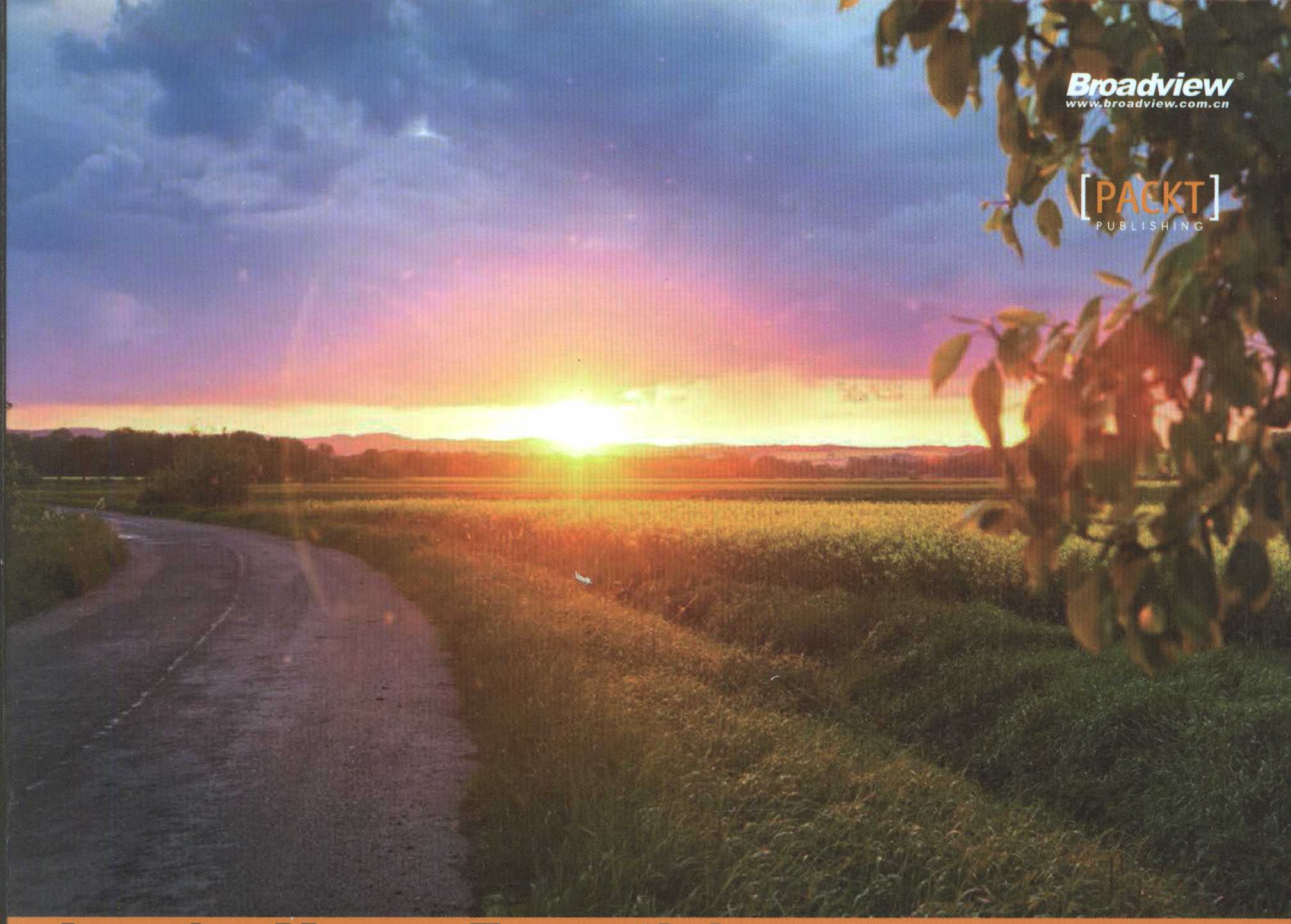


Broadview
www.broadview.com.cn

[PACKT]
PUBLISHING

Apache Mesos Essentials

Build and execute robust and scalable applications using Apache Mesos

Mesos

大数据资源调度与大规模容器运行最佳实践

[印度] Dharmesh Kakadia 著
DockOne社区：崔婧雯 刘梦馨 译

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Apache Mesos Essentials

Build and execute robust and scalable applications using Apache Mesos

Mesos

大数据资源调度与大规模容器运行最佳实践

[印度] Dharmesh Kakadia 著

DockOne社区：崔婧雯 刘梦馨 译

電子工業出版社

Publishing House of Electronics Industry

北京•BEIJING

内容简介

本书结合大量实例介绍了 Mesos 的使用方法、核心原理及框架开发的相关内容。通过这些内容读者可以在数据中心环境中利用 Mesos 搭建分布式系统、进行大数据分析及开发分布式应用。

全书分为8章分别从使用、开发和运维等角度全面展示了 Mesos 作为数据中心内核的强大能力、设计方面的精髓及在工程中的最佳实践。书中还介绍了 Mesos 项目的最新进展和未来的发展方向，并给出了大量参考文献和相关链接以方便读者进一步深入了解 Mesos。

本书适合分布式系统的研发、运维人员及相关技术爱好者阅读。

版权贸易合同登记号　图字：01-2015-5068

图书在版编目（CIP）数据

Mesos：大数据资源调度与大规模容器运行最佳实践/（印）卡卡迪亚著；崔婧雯，刘梦馨译. — 北京：电子工业出版社，2015.9
书名原文：Apache Mesos Essentials: Build and execute robust and scalable applications using Apache Mesos
ISBN 978-7-121-26902-8
I. ① M… II. ① 卡… ② 崔… ③ 刘…III. ① 数据处理软件 IV. ① TP274

中国版本图书馆CIP数据核字（2015）第 183823 号

策划编辑：张春雨
责任编辑：白　涛
印　　刷：北京中新伟业印刷有限公司
装　　订：北京中新伟业印刷有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路173信箱　　邮编 100036
开　　本：787×980 1/16　　印张：13　　字数：280 千字
版　　次：2015年9月第1版
印　　次：2015年12月第2次印刷
定　　价：65.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。

本书献给所有为之付出的人，也献给每一位 Mesos 爱好者。

推荐语

相比于容器生态圈的其他开源项目，Apache Mesos 绝对可以称得上是“前辈”，因为它在2013 年成为 Apache 的顶级项目之时，Docker 技术才刚刚开始发展。也正是由于 Docker 等容器技术的蓬勃发展，才让人们重新审视 Mesos 的价值。Mesos 目前已在 Twitter、Airbnb 等公司得到广泛应用，这也充分证明其已足够健壮，可以满足企业生产环境的需求。容器时代，它更被看作 Docker 的黄金搭档。

InfoQ 在很早之前就已经开始关注 Mesos 技术的发展，今年也陆续策划了相关专题，目的只有一个，那就是让容器技术在国内能够得到更好的应用，从而更好地推动业务发展。很高兴看到国内这么快就有了关于 Mesos 的中文书籍，我不得不感叹，这真是一个属于技术人的美好时代!

郭蕾　InfoQ主编

大数据时代的到来对数据存储和处理能力提出了新的要求，为了应对业务、数据量及时效性的需求，数据中心内往往存在复杂的应用程序和服务，比如批量计算、流式处理、多轮迭代式计算、各种在线服务等。如何对数据中心资源、作业服务进行统一的管理和调度成为了新的热点，在这种形势下，各种资源管理和调度系统应运而生，例如Borg（Google）、YARN（Apache Hadoop）、Matrix（百度）、Torca（腾讯）、Corona（Facebook）。Mesos 作为其中的优秀代表解决了两类问题，一是通过资源的统一管理和调度提升了利用率，二是通过自动化部署提升了服务运维效率。

《Mesos：大数据资源调度与大规模容器运行最佳实践》这本书深入浅出地介绍了 Mesos 的基本原理、API 框架及多种典型的解决方案，既有原理，又有应用案例。无论是站在服务开发者还是服务运维者的角度，读过此书后都会有一种醍醐灌顶的感觉，对 Mesos 有一种全新的认识，在服务管理、构建和运维上获得新的思路。

谢广军　百度公司高级技术经理，云计算技术服务人

Mesos 的故事始于 2009 年 UC Berkeley 的 AMP Lab，博士生 Benjamin Hindman、Andy Konwinski 和 Matei Zaharia 在研究集群资源的共享和调度时，创造了 Mesos，开创了“数据中心工程师”的新时代。如今，Mesos 已经在 Twitter、Airbnb 等国外互联网公司得到了大规模的实践应用，“分布式”、“微服务”、“容器”这些当下最热门的技术词汇中都可以看到 Mesos 的影子。

由 DockOne 社区翻译的《Mesos：大数据资源调度与大规模容器运行最佳实践》一书，作为国内第一本关于 Mesos 的书籍，从实践入手，将带你深入浅出地认识 Mesos，对国内 Mesos 实践者来说，是很好的入门读物。我公司的刘梦馨有幸参与了本次翻译工作，未来我们也会在社区和开发者的技术普及工作中贡献更多的力量。

左玥　灵雀云创始人兼CEO

现如今，数据中心承载了越来越多的复杂的数据处理业务，例如批量处理、流处理、图形分析和实时处理等。也正是如此多样化、差异化的数据业务推动着数据中心软件技术的不断演进。在众多主流数据中心操作系统中，Mesos 脱颖而出，成为了最热门的技术之一。其关键的技术特点在于它能够利用资源的高度抽象和两层调度机制，很好地解决多样化数据处理业务的资源管理和使用问题。

本书首先详细介绍了如何在 Mesos 上搭建、运行和调优 Hadoop、Spark 等技术的最佳实践，具有很强的操作性。在对基本使用有了一定了解后，作者深入剖析了 Mesos 的基本原理和开发、运维的技术细节。这是一本理论与实践相结合的实践性书籍，我相信无论是新手或是资深用户，都可以从中获得所需要的知识和技能。

汪洋　华为开源能力中心开源战略规划专家

互联网产品的推陈出新、快速迭代与海量访问、尖锋时刻等特点正在颠覆着传统的运维规划设计理念。从天、小时级响应加速到分钟、秒级响应，弹性计算的能力越来越大地影响着用户体验。同时互联网产品的开发、运营复杂程度已经不亚于汽车工业。对于开发工程师而言，如果可以不用过多关心分布式的问题而专注于代码实现将更有利于用户体验的提升。就在刚刚发布的 0.23.0 版本中，我们看到了 Mesos 的新特性更贴近业务场景，更接地气，而这些都充分说明了 Mesos 短短两年时间就成为 Apache 基金会明星项目的原因。如果你想了解 Mesos 的一切，如果你想与硅谷最新技术同行，请与我们一起认真阅读这本佳作。

涂彦　腾讯游戏运维总监

如何有效地调配与利用计算资源是IT技术人员始终需要面对的事，为此人们做了大量的实践和探索，而Apache Mesos项目给了我们一个很好的选择。本书覆盖了Mesos从实践到机理及扩展等多个阶段，一本书可窥项目全貌。更可贵的是本书译者都是资深的IT人，用精确的技术语言给予原书准确的翻译，这在引进技术书籍当中是难得的。

DockOne已经成为泛容器技术最大的技术社区。本次推动翻译此书，反映了其对容器这个技术方向上的认识，希望社区发展得更好，在新技术的推动和推广方面走得更远。

周东波　首都在线总工程师

微服务架构、轻量级组件、分布式设计、云端部署和运维是未来软件的趋势。软件如何在充分利用云平台能力的同时又避免平台锁定（Cloud Locked-In），软件如何以较低的成本实现互联网级别的高可用性并且应对爆发式流量？这是摆在开发人员面前的两大挑战。

近年来DCOS（数据中心操作系统）引起了越来越多开发者和运维工程师的关注，以Mesos、Kubernetes、CloudFoundry等为代表的DCOS，将有可能演化为应用和云平台之间的抽象层，也就是云时代的操作系统。这本Apache Mesos最佳实践，是了解和学习DCOS的优秀教材，两位译者在分布式领域实战经验丰富，译文通俗易懂，是一本不可多得的好书。

喻勇　DaoCloud联合创始人

Docker在短短两年时间内迅速蹿红，带动了周边软件生态圈的发展，其中Mesos + Docker就是最佳拍档的典范。纵观现在市面上竞争激烈的公有云来说，OpenStack + KVM确实是个很好的选择，但由此带来的统一计算Nova、统一网络Neutron、统一存储Gluster/Ceph的难度也是不容小觑的。而对于私有云来说，我们只想静静地以业务为驱动，把公司的资源做深度整合和统一调度，真正做到容器化、弹性可变化和可扩展。Docker的创新之处在于它的构建无论是对运维人员、开发人员还是测试人员都极其友好，统一Dockerfile，统一快速部署。当然，这就会遇到成百上千个容器如何运行管理、如何进行资源调度的问题，于是，基于Google Borg的开源实现Apache Mesos就是最好的解决方案。所以我相信对于私有云平台的构建，Mesos + Marathon + Docker会是一个很好的解决方案。如果你跟我一样为私有云的使用而苦恼，不妨试试Mesos，体验一个全新而有趣的思路，绝对会给你惊喜。

邵海杨　UPYUN联合创始人兼运维总监

Mesos 作为 DCOS 的发起者和推广者，完美地向我们展示了如何借助容器技术配合先进的资源管理与调度算法来完成对一个大规模机器集群的单机化抽象。不过这并不是关键。Mesos 的杀手锏其实在于两层调度和 framework，这两个特性使得 Mesos 的使用者可以通过插拔式的过程在 Mesos 之上构建各种各样的服务。所以，从大数据时代的不冷不热，再到容器时代的炙手可热，Mesos 用一个完美的翻身教会了我们“兼容并包”而非“一味竞争”才是平台级项目生存下去的看家本领。本书从实践到理论循序渐进，很好地展示了上述 Mesos 的核心功能和实现原理，是一本不可多得的国外技术书籍译本，值得一看。

张磊 博士　浙江大学计算机学院云计算团队科研人员

近年来，虚拟化的深入普及消灭了绝大部分自行建设、维护机房的需求。然而，大多数应用仍然停留在传统的单机模型上，无法充分发挥分布式云计算的真正威力。

书中介绍的 Mesos 是这样一个充满魔力的数据中心操作系统。它将整个集群抽象成一个大型资源池，托管在其中的应用可以自由伸缩、扩展，充分利用整个集群的资源。使用 Mesos 框架就像操作着《少数派报告》中精巧的 dashboard，控制着集群中几十上百个应用的部署、启动和结束任务，你只需动动手指，便可瞬间完成计算集群的重新配置。

在 Mesos 框架下，节点不再是一个个独立的个体，它们就像整齐划一的军队，随时接受着 Mesos 的调配。透过 Mesos 这样一个透视镜，你既可以站在几万英尺高的云端观看整个集群的微缩远景，又可以深入到集群中每个节点中的每个小任务里看其具体的执行情况，一切尽在掌握。

相信读过本书后，你也会迷上这个引领了云计算发展方向的 Mesos 框架!

孙宇聪　Coding CTO

*Apache Mesos Essentials*是一本出色的详细解读 Mesos 并具备实践运用的书籍，很高兴看到它这么快就在国内有了中文译本。Mesos 是 Apache 下的开源分布式资源管理框架，被称为分布式系统的内核。其特点在于资源管理和调度，能够消除集群硬件的差异化。在使用场景上，Mesos 可以作为资源池提供分配给上层的框架，同时支持多种用途的数据应用框架，比如 Hadoop、Kafka、Spark 等。近两年来以 Docker 为代表的容器技术在云计算行业大放异彩，而 Mesos 作为专门的集群管理器，是最适合大规模容器运行的调度框架之一，国内像时速云、灵雀云、数人科技等都已经将 Mesos 应用于实际的产品环境。本书籍不仅从理论上详细阐述了 Mesos，而且针对 Mesos 开发者和数据中心规模管理者提供了最佳实践，是一本不可多得的学习 Mesos 的书籍。

黄启功　时速云创始人

为提高物理机的使用率，广大运维同人曾经埋首于各种私有云技术，试图实现资源池化；但实施起来往往劳民伤财，非常痛苦，甚至无疾而终。Mesos 的出现，彻底改变了这个局面，它提供了一种更先进、更友好、更可控的资源池化技术，实现了真正意义上的数据中心操作系统。

本书实操性非常强，而且编排方式独特，首先直接阐述 Mesos 的各种应用场景，之后再深入讨论 Mesos 的基本原理和框架细节，并在最后贴心地附上 Mesos 运维的相关实践技术，适合运维同人作为 Mesos 工具书经常翻阅学习。

萧田国　高效运维技术社区创始人北京触控科技运维总监

大概在 2002 年的时候，曾经流行过网格计算的概念，大致就是把一组 PC 服务器组成资源池，通过资源调度最大化资源利用率并提高应用的可靠性。10 年之后，Mesos、Kubernetes 又流行起来，感觉昨日重现，这里面最重要的原因就是 Docker，没有 Docker 就不会有 Mesos 的流行，所以希望大家深入阅读此书。

赵鹏　Hyper 顾问

推荐序1

很高兴受 DockOne 社区的邀请，为*Apache Mesos Essentials*的中译版《Mesos：大数据资源调度与大规模容器运行最佳实践》作序。随着分布式计算技术在大数据处理方面的广泛应用，分布式集群已经成为企业数据中心的标准配置。分布式集群虽然性能优异、成本低廉，但是由于服务器数量众多，集群管理复杂度很高，只有 Google、Amazon、Facebook、Microsoft 之类的 IT 巨头能够建造和管理，而且这几家巨头的集群管理系统并不开源，不给其他企业使用。再者，云计算技术的兴起，极大地改变了企业 IT 系统的运作方式，从传统物理服务器搭建数据中心到软件定义数据中心。相应地，集群管理系统也从管理物理服务器向云时代操作系统转变。Apache Mesos 的出现，正好满足了广大企业对云时代数据中心管理的需求，Mesos 也正在逐渐向云时代操作系统演化。

Apache Mesos 是源自 UC Berkeley（加州大学伯克利分校）的分布式集群管理系统。Mesos 最早的发起人之一 Benjamin Hindman，当时还是 UC Berkeley 的计算机系博士生，他在 Google 实习的时候，受 Google 的分布式集群管理系统 Borg 的启发开始研发 Mesos。2010 年，Benjamin 在 Twitter 做了关于 Mesos 的讲座，然后 Twitter 逐渐开始使用 Mesos，最终成为 Mesos 最大规模的企业用户。在 Twitter，最大规模的 Mesos 集群有超过一万台服务器。硅谷的其他高科技公司，比如 Apple、Airbnb、Yelp、ebay 等，都在使用 Mesos来管理分布式集群。2013 年，Mesos 成为 Apache 顶级项目，同年 Benjamin 创立了 Mesosphere 公司，专门推广和商业化 Mesos。2014 年，首届 Mesos 大会在芝加哥召开，有二百多人参会。2015 年，Mesos 大会将在西雅图召开，预计有近千人参加。

Mesos 为什么在企业有很成功的大规模应用？个人觉得这主要得益于 Mesos 简单易用。分布式集群管理是个很复杂的系统，比如 Google 的 Borg，功能很强大也很复杂，Borg 管理了 Google 数千万台服务器，其复杂程度可想而知。但是 Mesos 的理念是要把分布式集群管理系统变得简单易用。Mesos 采用了两级调度算法来管理分布式集群。两级调度指的是资源分配和任务调度。Mesos 本身只负责资源分配这一件事（Mesos 本来也有任务隔离功能，在 Docker 出来以后，人们更倾向于用 Docker 来做任务隔离），Mesos 本身并不做任务调度，

而是把任务调度的功能交给基于 Mesos 之上的各种 framework 来完成。比如，Mesos 之上可以运行 Hadoop、Spark 之类的大数据平台，Mesos 只负责将集群资源分配给 Hadoop、Spark 任务，然后 Hadoop、Spark 负责调度各自的任务，如申请任务资源、重启失败任务等。Mesos 这样设计，降低了集群管理系统复杂度，进而降低了开发难度，方便 Mesos 作为开源项目来开发维护。毕竟开源项目不是商业软件，开源项目如果设计复杂度很高，会使得项目开发捉襟见肘，项目质量也很难得到保证。

近来 Mesos 很流行的一个重要原因是 Mesos 对 Docker 的原生支持。Docker 自 2013 年开源以后，发展得如火如荼，已基本成为新一代应用发布标准。2014 年下半年，Mesos 发布 0.20 版本开始原生支持 Docker，这样一来，Docker 任务可以很方便地在 Mesos 管理的集群上运行。于是 Docker 公司的创始人兼 CTO，Solomon Hykes，在 2014 年年底欧洲的 Docker 大会上说："Mesos 是在大规模集群生产环境中运行 Docker 的黄金搭档。"可以说，Docker 的出现简化了开发人员对应用程序的测试和交付工作，Mesos 加上 Docker，又极大地方便了企业在生产环境中大规模使用 Docker。Mesos 和 Docker 互相推波助澜，会深刻改变未来企业 IT 系统。

Mesos 另外一个优点是开放性，采用 Mesos 来管理集群并不意味着就排除了使用其他集群管理系统的可能性。Mesos 的核心是分布式集群的资源分配，不负责任务调度，因而 Mesos 可以和其他集群管理系统协同工作。比如 Google 开发的 Kubernetes，Docker 公司推出的 Swarm 等，这些面向 Docker 的集群管理系统都可以运行在 Mesos 之上，成为 Mesos 的 framework，Mesos 为 Kubernetes 或 Swarm 调度 Docker 任务分配资源。企业如果已经采用了 Mesos 来管理其集群，还可以采用 Kubernetes 或 Swarm 来做任务调度，充分利用不同集群管理系统的特性。

当然 Mesos 目前处于比较早期的阶段，还不能说非常成熟。Mesos 主要实现了分布式集群管理系统中的资源分配功能，其他功能主要靠额外的组件来实现，比如服务发现和负载均衡用 Marathon 加 HAproxy 来实现等。再者，Mesos 对于存储管理还没有很成熟的解决方案，对于多租户管理也不够强大。

最后，我对 Mesos 的发展做出一点展望。随着云计算技术的蓬勃发展，企业的数据中心都将移到公有云或私有云上，未来 Mesos 会逐步演化为云时代的操作系统。有了云时代操作系统，企业使用整个数据中心就像使用一台服务器一样简便易行，再也不用操心数据中心的管理运维，只需要专注于发展业务。进而，所有企业级软件都将通过云操作系统来分发，不论是在公有云或私有云上，都能做到一键部署、免维护、免升级等，真正把企业级软件变成 SaaS 模式。

数人科技创始人　王璞

推荐序2

于此，我推荐本书给您，希望它成为您的千里马，陪伴您征服事业的最颠峰。如果 Docker 容器像海运业的集装箱（即容器），则 Mesos 集群就像一座巨大的现代化港口或码头。所以专家们常称赞：Docker 和 Mesos 是一长串绝佳搭档的最新组合，就像花生酱和果冻、牛奶和饼干那样完美。例如，Mesos 已经是一个相当稳定的平台了，并且可以使用 Docker 镜像来部署形形色色的 Mesos 应用。

由于两者是力与美的最先进组合，对于新一代云平台的开发者或维运人员而言，孰悉 Mesos 的架构、程序及管理模式，将是彻底掌握 Docker + Mesos 的最佳途径。

在这本书里，原作者从现代基础架构对数据中心操作系统的需求出发，阐述了 Mesos 如何为企业大规模运行容器提供最实用的方法，让企业有信心克服一连串关于可用性、规模和性能等方面非常深层次的挑战。例如，第 2、第 3 章深入浅出地探讨了如何在 Mesos 上使用 Hadoop 处理大数据及如何调优 Mesos 上的 Spark。

俗语说：众星拱月。虽然 Docker 是一弯明亮的月儿，但在 Mesos 平台上还可结合许多其他的架构和服务，例如，第 4、第 5 章演示了在 Mesos 上部署 lambda 架构，并详细讨论了 Marathon、Chronos 和 Aurora 框架，帮助大家理解如何在 Mesos 上部署这些服务。

当你孰悉了上述 Mesos 架构之后，即可深入理解这一平台的实现、开发和运维的细节了。理解在大规模使用场合，如何实现高度自动化的环境，跨成百上千台服务器集群下运行容器和处理大数据。这些都是你所需要的关键技术。因此，本书后段的第 6、第 7、第 8 章详细介绍 Mesos 的资源限制、隔离等管理策略，Mesos 框架 API，以及 Mesos 平台的运维细节。这些都能帮你建立扎实的根基并获得丰富的经验。

刚才已经提到了，Mesos + Docker 是一个力与美的最佳组合。一旦你用心学完本书的内容，就掌握了大规模运行 Docker 容器的基本观念和使用方法了。随着近年来 Docker 集装箱技术的流行，容器式软件设计思维已迅速蔚为风潮，正在强力改变终端与云平台的整体设计理念和方法。这跟半世纪前，海上船运业的集装箱风潮，对整个产业的革命性改变，其力

道来源和所产生的经济效益，可说具有异曲同工之妙。因之，本书将是你登上这一革命性技术潮流之巅的最佳垫脚石，也是你的幸运草，将给你或你的企业带来可观的商业利益，也为你带来新的机遇。

一旦掌握了 Mesos 和 Docker 的知识，拥有了云平台的运维和管理技能，你就能强力支持互联网和大数据的各种应用系统开发了。在需求、数据和软件愈来愈碎片化的趋势下，容器包装微服务成为主流。这些微服务在运行时间（Run-time）经常需要动态组合成为各式各样的应用来支撑企业多变的业务流程。此时，你就有足够的能力去促进终端与云平台的一致性设计，有效协助创造业务应用系统之间极为迅速的、瞬间的动态组合，来支撑企业流程和活动，以创新模式将它们组合起来，创造端云整合的最佳用户体验。

台湾 Docker 论坛主席　高焕堂

关于作者

Dharmesh Kakadia 是微软研究院的研究员，负责开发下一代集群管理系统。在加入微软研究院之前，他在海得拉巴信息技术国际研究所获得硕士学位，致力于改进云和大数据系统的任务调度。他对系统和数据的交集部分非常感兴趣，在资源管理领域发表了多篇研究报告。他热衷于开源技术，活跃在多个开源社区里。他的 Twitter 账号是 `@DharmeshKakadia`。

我要感谢一直支持我的家人、朋友和同事。也要感谢本书的审校者和 Packt 出版社的所有工作人员，是大家共同的不懈努力才确保本书按时保质地完成。没有大家的帮助，就没有本书的出版。

关于审校者

Tomas Barton 是布拉格市捷克技术大学的在读博士，他专注于分布式计算、数据挖掘和机器学习。他从 Mesos 的早期版本发布时就开始使用它。他也为 Debian 打包贡献代码，并维护 Mesos 自动化安装的 Puppet 模块。

Andrea Mostosi 是一位技术热爱者，从小热爱创新。从 2003 年开始工作，他在多个项目里尝试了软件科学领域几乎所有的职位。目前是 Fool 公司的 CTO，这是一家尝试挖掘 Web 和社交数据的公司。空闲时他喜欢旅行、跑步、烹饪、骑车和写代码。

> 我要感谢极客朋友 Simone M、Daniele V、Luca T、Luigi P、Michele N、Luca O、Luca B、Diego C 和 Fabio B。他们是我认识的最聪明的人，和他们在一起总能促进我追求卓越。

Sai Warang 是加拿大创业公司 Shopify 的软件开发工程师。他目前致力于构建在成千上万网络交易中防止欺诈的实时工具。之前，他在滑铁卢大学学习计算机科学，在旧金山的 Tagger 和 Zynga 的多个数据分析项目中工作过。他偶尔也会涉猎创作写作。

前言

Mesos 使得开发并管理自动容错且可扩展的分布式应用程序变得很容易。它支持为聚合的资源池编程，而无须考虑单台机器的资源管理。使用 Mesos 后，你心仪的所有框架，从长时间运行服务的数据处理到 Web 服务的数据存储，都可以在同一个集群里共享资源。Mesos 的一致性基础架构及容错性也能简化大规模部署的运维工作。运行 Mesos 时，单点故障不会影响到应用程序的持续运行。

使用 Mesos，每个人都能开发分布式应用程序，并轻松将其扩展到成千上万的节点上。

本书范围

第 1 章解释现代基础架构对数据中心操作系统的需求，以及 Mesos 为何是满足这一需求的绝佳选择；还介绍了在各种环境里，如何安装单节点和多节点的 Mesos。

第 2 章探讨了如何在 Mesos 上使用 Hadoop 进行批量数据处理。

第 3 章讲述了如何在 Mesos 上运行 Spark，及如何调优 Mesos 上的 Spark。

第 4 章演示了 Mesos 上部署 lambda 架构的多种选择，详细讨论了 Mesos 上 Storm、Spark Streaming 和 Cassandra 的搭建。

第 5 章介绍了服务及在 Mesos 上服务架构的不同之处，详细讨论了 Marathon、Chronos 和 Aurora 框架，帮助大家理解如何在 Mesos 上部署这些服务。

第 6 章深入介绍了 Mesos 基本原理，详细介绍 Mesos 的资源分配、隔离和容错的实现细节。

第 7 章讲述了在 Mesos 上框架开发的细节，以及通过构建 Mesos 框架学习 Mesos API。

第 8 章探讨 Mesos 的运维细节，讨论了监控、多租户、可用性和维护各方面及 REST API 和配置细节。

本书要求

理解本书内容，需要熟悉 Linux，并且具备基本的编程知识。此外，由于 Mesos 的分布式特性，如能够访问多台机器或者某个云服务，将有助于加深对本书内容的理解。

本书面向的读者

本书适合所有想要使用 Mesos 开发和管理数据中心规模的应用程序的人。

排版约定

本书为区分不同类型的信息，采用了多种文本风格。下面给出这些风格的示例，并解释其含义。

代码、文件夹名、文件名、路径名和配置参数如下：“仓库包含的 vagrant 文件和 README 文件会提供更多详细信息。”

代码段如下：

```
<property>
    <name>mapred.mesos.framework.secretfile</name>
    <value>/location/secretfile</value>
</property>
```

命令行的输入输出如下：

```
ubuntu@local:~/mesos/ec2 $ ./mesos-ec2 destroy ec2-test
```

新概念和**重要的单词**会加粗显示。屏幕上能看到的单词，比如，菜单或对话框上的单词，会这样显示：“在 Web UI 上，点击 **+New Job**，会弹出仪表板显示作业的详细信息。”

警告或重要注意事项会放在这样的框里

提示和技巧这么显示

读者反馈

欢迎读者的反馈。让我们知道你们对本书的看法，喜欢哪一部分，不喜欢哪一部分。读者的反馈对我们而言十分重要，因为这些反馈将帮助我们开发出读者真正喜欢的书籍。

请将反馈内容发送到邮箱 feedback@packtpub.com，记得在邮件主题中写上对应的书名。

如果你是某领域的专家，有兴趣撰写或者出版书籍，请查看 www.packtpub.com/authors 的作者指南。

客户支持

对于 Packt 出版社的尊贵读者，我们提供了很多途径来最大化读者利益。

下载样例代码

可以使用 Packt 出版社账户从 http://www.packtpub.com 下载你所购买的所有 Packt 书籍的样例代码文件。如果是通过非官网途径购买的，可以访问 http://www.packtpub.com/support 进行注册，我们会将文件直接用邮件发送给你。

下载书籍的彩色图像

我们还提供PDF文件，包含书中在截图或图表里列出的所有彩色图片。这些彩色图片有助于更好地理解书籍的内容，可以从 https://www.packtpub.com/sites/default/files/downloads/8762OS_ColoredImages.pdf 下载该文件。

勘误表

虽然我们已经尽力保证本书内容的准确性，但还是会有一些错误。如果你找到本书的错误之处——可能是文本或代码的错误——请告知。这样不仅可以帮到其他读者，还能帮助我们改进本书的后续版本。无论发现任何错误，都可以访问 http://www.packtpub.com/submit-errata，选择该书籍，点击“**Errata Submission Form**”链接，填写需要勘误的详细信息。一旦勘误被验证，提交就会被接受，勘误表会上传到我们的网站上或添加到该书的勘误部分。

在 https://www.packtpub.com/books/content/support 的搜索字段输入书籍名称，

可以查看之前提交的勘误信息。所需信息会列在 **Errata** 部分。

盗版

网络上通过各种媒体进行的盗版行为日益严重。Packt 出版社严肃对待版权和许可证问题。如果你无意中在网络上发现我们的出版物被以任何形式非法复制，请立即告诉我们地址或网站名称，以便及时补救。

请联系 `copyright@packtpub.com` 报告疑似盗版材料的链接。

非常感谢帮助我们保护作者权益，这样我们才能一起创造更多有价值的内容。

问题

如果对本文的任何内容有疑问，请联系 `questions@packtpub.com`，我们一定尽力解决所有问题。

目录

第1章 运行 Mesos

本章简要介绍 Apache Mesos 和集群计算框架，并会逐步讲解如何在单节点和多节点上搭建 Mesos。还会讲述如何使用 Vagrant 在 Amazon EC2 上搭建 Mesos 集群。本书通篇会交替使用 Apache Mesos 和 Mesos 两个术语。本章包括如下内容：

- 现代数据中心
- 集群计算框架
- Mesos 简介
- 为什么选择 Mesos
- 单节点 Mesos 集群
- 多节点 Mesos 集群
- Amazon EC2 上的 Mesos 集群
- 使用 Vagrant 运行 Mesos
- Mesos 社区

现代数据中心

现代应用程序高度依赖于数据。企业生成并处理的数据呈指数级增长，这逐渐改变了我们存储及处理数据的方式。当为存储和处理数据规划现代基础架构时，已经无法仅仅通过购买硬件扩容来解决问题了。批量处理、流处理、面向用户服务、图像处理及实时分析，这些不同的框架变得和支撑它们运行的硬件同样重要。这些框架才是数据中心世界里的支柱

应用程序。

大数据的规模和多样性导致，对于现代工作负载而言，传统的扩容策略已经落伍。因此，大型企业转向分布式处理，把大量计算机当成单个巨型机器来使用。很多资源需求各异的应用程序共享集群，多种架构之间高效共享资源的关键是力争达到资源的高利用率。需要将所有的小型机器整合成单个大型计算机。Mesos 天生就是这些计算机集群的核心。

传统做法是，各种框架分别单独运行，在框架间静态划分资源，这样会导致资源的使用效率很低。将大量日常机器当作单个大型机器使用，能够在所有框架之间弹性地共享资源，这些需求都要求集群计算框架。Mesos 的灵感来源于集群内多个框架之间共享资源和提供资源隔离性的想法。

集群计算框架

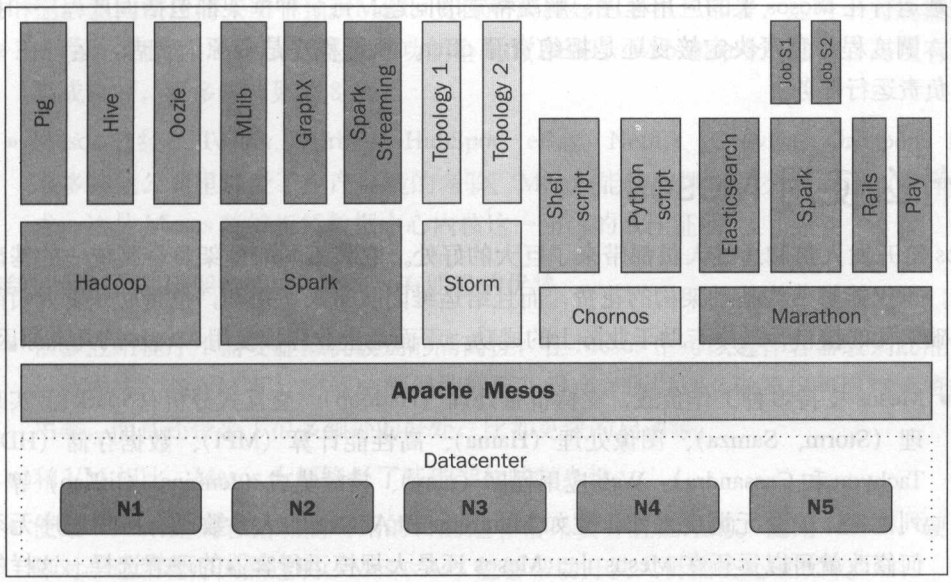
在现代集群里，不同的框架所要求的计算需求会非常不同，企业需要运行多种框架，并在其间共享数据和资源。资源管理程序面临巨大的挑战和互为矛盾的目标：

- 高效性：高效共享资源是集群管理软件的终极目标。
- 隔离性：当多个任务共享资源时，最重要的考量之一是确保资源的隔离性。隔离性和正确调度的整合是保证服务级别协议（SLA）的基础。
- 可伸缩性：现代基础架构的持续增长要求集群管理程序可以线性伸缩。一个重要的可伸缩性指标是框架伸缩决策制定所需的延时。
- 健壮性：集群管理是中央组件，持续的业务运营要求健壮的集群管理。从良好测试的代码到容错设计，很多方面都有助于提高健壮性。
- 可扩展性：在任何公司里，集群管理软件的开发量都非常大，而且这些软件可能已经使用了数十年。运营期间，企业政策和/或硬件的变化都不可避免地要求集群资源管理方式的改变。因此，对于大型企业来说，可维护性非常重要。集群管理软件必须是可配置的，同时考虑到很多约束条件（比如位置、硬件等），并且能够支持多种框架。

Mesos 简介

Mesos 是集群管理器，力争通过在多种框架之间动态共享资源来优化资源使用率。该项目于 2009 年由位于 Berkeley 的加利福尼亚大学发起，已经在很多公司的生产环境上使用过，包括 Twitter 和 Airbnb。2013 年 7 月，在该项目孵化大概两年时，就成为 Apache 的最高级别项目。

Mesos 在多种不同类型的工作之间共享机器（或者节点）的可用资源，如下图所示。Mesos 可以看作是数据中心的内核，提供所有节点资源的统一视图，所起的作用类似于操作系统内核在单台机器上的作用，可以无缝地访问多节点资源。Mesos 提供了帮助构建数据中心应用程序的核心程序，其核心组件是可伸缩的两阶段调度程序。Mesos API 允许访问各种不同的应用程序，而无须向 Mesos 核心程序提供特定领域的信息。因为关注于内核，Mesos 不会遇到中央式调度器天然存在的各种问题。



Mesos 作为数据中心内核

下列组件对于理解 Mesos 的整体架构至关重要。本节会做简要介绍，之后第 6 章会更加详细地探讨整体架构的细节。

master

master 负责在 slave 资源和框架之间进行调度。任何时候，Mesos 只有一个活动的 master，由 ZooKeeper 使用分布式一致性算法选举产生。如果 Mesos 配置运行在容错模式下，会通过分布式主选举协议选出一个 master，其他 slave 则处在待命模式。从设计角度来讲，Mesos master 本身不会用来做任何重负载计算，这样简化了 master 的设计。它以资源 offer 的形式将 slave 的资源提供给框架，并且根据已接受的 offer 在 slave 上启动任务。它同时也负责任务和框架之间的所有通信。

slave

slave 是 Mesos 集群里真正工作的节点。它们管理单个节点上的资源，遵守资源政策来适应业务优先级。slave 管理各种资源，比如 CPU、内存、端口等，同时执行框架递交的任务。

框架

框架是运行在 Mesos 上的应用程序，解决特定的问题。每一种框架都包括调度程序和执行程序。调度程序负责决定接受还是拒绝资源 offer。执行程序是资源消费者，运行在 slave 上，负责运行任务。

为什么使用 Mesos

Mesos 给开发人员和运维人员都带来了巨大的好处。它将不同的框架整合到统一的基础框架上，不仅能够节约基础架构的花费，而且给运维团队带来了便利，也帮助开发人员简化了基础架构的接口，最终有助于业务上的成功。下面是企业应该使用 Mesos 的几点原因：

- Mesos 支持多种工作负载，支持批量处理（Hadoop）、交互式分析（Spark）、实时处理（Storm、Samza）、图像处理（Hama）、高性能计算（MPI）、数据存储（HDFS、Tachyon 和 Cassandra）、Web 应用程序（play）、持续集成（Jenkins、GitLab）等一系列框架。而且元调度框架，比如 Marathon 和 Aurora 的大多数已有应用程序无须任何修改就可以运行在 Mesos 上。Mesos 还是大规模运行容器的理想选择。这样的灵活性使得 Mesos 非常易于部署。
- Mesos 通过在框架间弹性共享资源来改进资源使用率。如果没有统一的数据中心操作系统，不同框架就只能运行在隔离的硬件上。这样的资源静态划分会导致资源碎片化，限制资源使用率和吞吐量。通过 Mesos 实现资源的动态共享，能够帮助达到更高的资源使用率和吞吐量。
- Mesos 是一个拥有活跃社区的开源项目。Mesos 的可插拔式架构使其可以针对企业需求灵活定制。Mesos 可以在很多不同的操作系统和硬件上运行，给使用者提供了非常丰富的选择，避免依赖于某个特定的供应商。因此，基于Mesos API 开发的程序可以在多种基础架构上运行。也就是说，Mesos 应用程序可以同时运行在物理机、虚拟基础架构和云提供商上。
- Mesos 最重要的优势应该是帮助开发人员构建高效的现代应用程序。开发人员从为单台机器开发应用程序转变成为数据中心开发程序，他们需要 API 帮助其关注于自己的应用逻辑，而不用花时间理解分布式基础框架的细节。使用 Mesos，开发人员

无须关心分布式细节，只需关注自己应用程序特定领域的逻辑。同时，Mesos 提供了丰富的 API，帮助开发出可伸缩的，自动容错的分布式应用程序，更多细节见第 7 章。

- 大型基础架构的运维通常都十分困难。Mesos 通过提供资源的一致视图来简化基础架构的管理。这增加了运维的敏捷性，同时因为不需要分配单独的集群，用 Mesos 部署新服务速度会更快。Mesos 非常适合运维人员，它将基础架构资源当作整体管理，而不是一台台单独的机器。这意味着 Mesos 可以自适应故障，自动保证高可用性，而无须任何人工干预。Mesos 支持隔离良好的多租户部署，这对于大规模运营来说至关重要。Mesos 还提供了功能全面的 REST、Web 和命令行接口，与现有工具集成良好，更多细节见第 8 章。
- Mesos 已经在 Twitter、Airbnb、HubSpot、eBay、Netflix、Conviva、Groupon，以及很多其他公司里经受了生产环境的考验。Mesos 能够支撑不同类型公司的多样化需求，这是 Mesos 能够担任数据中心内核这一角色的最佳证明。

和传统的虚拟化基础架构相比，Mesos 有明显的优势：

- 很多应用程序不需要虚拟机级别的隔离性，在 Mesos 中可以选择容器级别的隔离来运行程序。与 VM 相比容器的额外消耗要小得多，所以这么做不仅可以增大资源使用率，而且还带来了很多额外的好处，比如更快的启动等。
- 和 VM 相比，Mesos 大幅降低了基础架构的复杂性。
- 使用 VM 来达到容错和高可用的目的是非常昂贵并且困难的。使用 Mesos，硬件故障对于应用程序天然透明，Mesos API 能够帮助开发人员轻松处理故障。

既然我们已经了解了使用 Mesos 的优势，那么就来创建一个单节点的 Mesos 集群，开始 Mesos 的探索之旅吧。

单节点 Mesos 集群

Mesos 可以在 Linux 和 Mac OS X 上运行。尝试使用 Mesos 的最简方式是搭建一台单机器的 Mesos，我们就从这里开始。目前，Mesos 并没有为不同的操作系统提供二进制包，需要使用者下载源代码进行编译。社区里有编译好的可用二进制包。

Mac OS

Homebrew 是 Mac 上 Linux 风格的软件包管理器。Homebrew 支持 Mesos 的安装，在本地进行编译。Mac 上安装 Mesos 需要如下步骤：

1. 安装 Homebrew，从 `http://brew.sh/` 下载。
2. Homebrew 要求安装 Java。Mac 自带 Java，因此只需确保 `JAVA_HOME` 设置正确。
3. 使用 Homebrew 安装 Mesos，键入如下命令：

```
mac@master:~ $ brew install mesos
```

虽然 Homebrew 提供了在 Mac 上搭建 Mesos 的方式，不过生产环境还是推荐在 Linux 上运行。

Fedora

从 Fedora 21 开始，Fedora 仓库里包含 Mesos 软件包。其中，mesos-master 和 mesos-slave 软件包分别安装在 master 和 slave 上。mesos 软件包包含 master 和 slave 的软件程序。在版本 ¿=21 的 Fedora 上安装 mesos 软件包，使用如下命令：

```
fedora@master:~ $ sudo yum install -y mesos
```

现在就可以按照之后“启动 Mesos”小节的步骤来运行 Mesos。对于版本小于 21 的 Fedora，需要安装依赖软件包，然后编译源代码后进行安装，这种方式类似于 CentOS，会在后文中详细介绍。

安装依赖软件包

Mesos 要求安装如下依赖软件包：

- g++（>=4.1）
- Python 2.6 开发包
- Java 开发工具集（>=1.6）和 Maven
- cURL 库
- SVN 开发库
- Apache 便携运行时库（APRL）
- 简单身份验证和安全层（SASL）库

另外，如果想从 git 库构建 Mesos 的话，还需要 autoconf（版本 1.12）和 libtool。Mesos 的安装在不同操作系统上有些区别，我们会详细介绍在 Ubuntu 14.10 和 CentOS 6.5 上安装 Mesos 的具体步骤。在其他操作系统上的步骤与之类似。

CentOS

按照下述步骤在 CentOS 上安装所需依赖软件包：

1. 目前，CentOS 默认仓库不提供 >=1.8 的 SVN 库，需要手动添加所需仓库。在 /etc/yum.repos.d/ 下创建文件 wandisco-svn.repo，并添加如下行：

```
centos@master:~ $ sudo vim /etc/yum.repos.d/wandisco-svn.repo

[WandiscoSVN]
name=Wandisco SVN Repo
baseurl=http://opensource.wandisco.com/centos/6/svn-1.8/RPMS/$base
arch/
enabled=1
gpgcheck=0
```

然后使用如下命令安装 libsvn：

```
centos@master:~ $ sudo yum groupinstall -y "Development Tools"
```

2. Maven 的安装需要先下载，然后解压缩，并且放到 PATH 里。如下命令会在下载完成后将文件解压缩到 /opt 目录，并将 mvn 链接到 /usr/bin:

```
centos@master:~ $ wget http://mirror.nexcess.net/apache/maven/maven
-3/3.0.5/binaries/apache-maven-3.0.5-bin.tar.gz
centos@master:~ $ sudo tar -zxf apache-maven-3.0.5-bin.tar.gz -C
/opt/
centos@master:~ $ sudo ln -s /opt/apache-maven-3.0.5/bin/mvn
/usr/bin/mvn
```

3. 使用如下命令安装其他依赖软件包：

```
centos@master:~ $ sudo yum install -y python-devel
java-1.7.0-openjdk-devel zlib-devel libcurl-devel openssl-devel
cyrus-sasl-devel cyrus-sasl-md5 apr-devel subversion-devel
```

Ubuntu

使用如下命令在 Ubuntu 上安装所需依赖软件包：

```
ubuntu@master:~ $ sudo apt-get -y install build-essential
openjdk-6-jdk python-dev python-boto libcurl4-nss-dev
libsasl2-dev libapr1-dev libsvn-dev maven
```

构建 Mesos

一旦所需软件都安装完成，就可以按照如下步骤构建 Mesos：

1. 从 `http://mesos.apache.org/downloads/` 下载最新的稳定版本。撰写本书时，最新版本是 0.21.0。将文件 `mesos-0.21.0.tar.gz` 保存到某个目录下。打开终端，进入之前保存文件的目录，或者直接在终端运行如下命令下载 Mesos：

   ```
   ubuntu@master:~$ wget http://www.apache.org/dist/mesos/0.21.0/mesos
   -0.21.0.tar.gz
   ```

2. 使用如下命令解压缩 Mesos，并进入解压缩后的目录。注意第二个命令会删除下载的 `.tar` 文件，并将解压缩的目录重命名，去掉版本号。

   ```
   ubuntu@master:~ $ tar -xzf mesos-*.tar.gz
   ubuntu@master:~ $ rm mesos-*.tar.gz ; mv mesos-* mesos
   ubuntu@master:~ $ cd mesos
   ```

3. 创建 `build` 目录。该目录会包含编译后的 Mesos 二进制文件。该步骤是可选的，但是推荐执行。这样可以将 build 出的文件发布到各个 slave，而无须在每个 slave 上都编译一次：

   ```
   ubuntu@master:~/mesos $ mkdir build
   ubuntu@master:~/mesos $ cd build
   ```

4. 运行 `configure` 脚本配置安装：

   ```
   ubuntu@master:~/mesos/build $ ../configure
   ```

   `configure` 脚本支持构建环境的调优，可以运行 `configure --help` 查看详细参数列表。缺失了任何依赖的软件包，configure 脚本都会报告出来，用户可以回头重新安装缺失的软件包。一旦配置成功，就可以进入下一步。

5. 使用 `make` 完成编译。这一步可能会花一些时间。第二个命令是 `make check`：

```
ubuntu@master:~/mesos/build $ make
ubuntu@master:~/mesos/build $ make check
```

make check 会构建出示例框架。现在就已经可以在 build 目录下直接运行 Mesos 而无须进行安装了。

6. 使用如下命令安装 Mesos：

```
ubuntu@master:~/mesos/build $ make install
```

下表是 Mesos 提供的一系列命令：

命令	用法
mesos-local.sh	该命令在单进程里启动驻内存的集群
mesos-tests.sh	该命令运行 Mesos 的测试用例套件
mesos.sh	这是启动 Mesos 命令的封装脚本。不带任何参数运行时会显示所有可用参数
gdb-mesos-*	该命令使用 gdb 以调试模式启动相关进程
lldb-mesos-*	该命令使用 lldb 以调试模式启动相关进程
valgrind-mesos-*	该命令启动相关 Valgrind instrumentation 框架
mesos-daemon.sh	该命令启动/停止 Mesos 守护进程
mesos-start-cluster.sh mesos-stop-cluster.sh	该命令启动/停止 Mesos 集群，集群节点定义在文件 [install-prefix]/var/mesos/deploy/masters 和 [install-prefix]/var/mesos/deploy/slaves 里
mesos-start-masters.sh mesos-stop-masters.sh	该命令启动/停止定义在 master 文件里各节点上的 Mesos master 程序
mesos-start-slaves.sh mesos-stop-slaves.sh	该命令启动/停止定义在 slave 文件里的各节点上的 Mesos slave 程序

安装完成后，就可以使用 mesos-local 命令启动本地 Mesos 集群，该命令会在单个进程里同时启动 master 和 slave 程序，可以帮助快速检查 Mesos 是否正确安装。

启动 Mesos

启动 Mesos 进程。首先，需要为 Mesos replicated log 创建目录，并赋予目录读写权限：

```
ubuntu@master:~ $ sudo mkdir -p /var/lib/mesos
ubuntu@master:~ $ sudo chown `whoami` /var/lib/mesos
```

如下命令会使用上面创建的目录，启动 master：

```
ubuntu@master:~ $ mesos-master --work_dir=/var/lib/mesos
I1228 07:29:16.367847 2900 main.cpp:167] Build: 2014-12-26 06:31:26 by
ubuntu
I1228 07:29:16.368180 2900 main.cpp:169] Version: 0.21.0
I1228 07:29:16.387505 2900 leveldb.cpp:176] Opened db in 19.050311ms
I1228 07:29:16.390425 2900 leveldb.cpp:183] Compacted db in 2.731972ms
...
I1228 07:29:16.474812 2900 main.cpp:292] Starting Mesos master
...
I1228 07:29:16.488203 2904 master.cpp:318] Master 20141228-072916-2517893
22-5050-2900 (ubuntu-master) started on master:5050
...
I1228 07:29:16.510967 2903 master.cpp:1263] The newly elected leader is
master@master:5050 with id 20141228-072916-251789322-5050-2900
I1228 07:29:16.511157 2903 master.cpp:1276] Elected as the leading master!
...
```

上面的输出里显示了 build 版本、master 使用的各种配置及集群里的 master ID。slave 程序应该能连上 master。slave 程序可以通过 --master 参数指定 master 的 IP 地址或者主机名。本书假定运行 master 的机器主机名为 master，读者实际操作时需要根据机器情况使用正确的主机名或者 IP 地址。

```
ubuntu@master:~ $ mesos-slave --master=master:5050
I1228 07:33:32.415714 4654 main.cpp:142] Build: 2014-12-26 06:31:26 by
vagrant
I1228 07:33:32.415992 4654 main.cpp:144] Version: 0.21.0
I1228 07:33:32.416199 4654 containerizer.cpp:100] Using isolation: posix/
cpu,posix/mem
I1228 07:33:32.443282 4654 main.cpp:165] Starting Mesos slave
```

```
I1228 07:33:32.447244 4654 slave.cpp:169] Slave started on 1)@ master:5051
I1228 07:33:32.448254 4654 slave.cpp:289] Slave resources: cpus(*):2;
mem(*):1961; disk(*):35164; ports(*):[31000-32000]
I1228 07:33:32.448619 4654 slave.cpp:318] Slave hostname: master
I1228 07:33:32.462025 4655 slave.cpp:602] New master detected at master@
master5050
...
```

如上输出证明 slave 已经连接上了 master，并且列出了 slave 资源。至此，集群启动成功，已经有一个运行着的 slave 以供随时运行框架。

运行测试框架

Mesos 包含了多种使用 C++、Java 和 Python 编写的示例测试框架。这些框架可以用来验证集群是否配置正确。如下测试框架使用 C++ 运行五个示例应用。使用如下命令运行：

```
ubuntu@master:~/mesos/build/src $ ./test-framework --master=master:5050
I1228 08:53:13.303910 6044 sched.cpp:137] Version: 0.21.0
I1228 08:53:13.312556 6065 sched.cpp:234] New master detected at master@
master:5050
I1228 08:53:13.313287 6065 sched.cpp:242] No credentials provided.
Attempting to register without authentication
I1228 08:53:13.316956 6061 sched.cpp:408] Framework registered with
20141228-085231-251789322-5050-5407-0001
Registered!
Received offer 20141228-085231-251789322-5050-5407-O3 with mem(*):1961;
disk(*):35164; ports(*):[31000-32000]; cpus(*):2
Launching task 0 using offer 20141228-085231-251789322-5050-5407-O3
Launching task 1 using offer 20141228-085231-251789322-5050-5407-O3
Task 0 is in state TASK_RUNNING
Task 0 is in state TASK_FINISHED
Task 1 is in state TASK_RUNNING
Task 1 is in state TASK_FINISHED
Received offer 20141228-085231-251789322-5050-5407-O4 with mem(*):1961;
disk(*):35164; ports(*):[31000-32000]; cpus(*):2
Launching task 2 using offer 20141228-085231-251789322-5050-5407-O4
```

```
Launching task 3 using offer 20141228-085231-251789322-5050-5407-O4
Task 2 is in state TASK_RUNNING
Task 2 is in state TASK_FINISHED
Task 3 is in state TASK_RUNNING
Task 3 is in state TASK_FINISHED
Received offer 20141228-085231-251789322-5050-5407-O5 with mem(*):1961;
disk(*):35164; ports(*):[31000-32000]; cpus(*):2
Launching task 4 using offer 20141228-085231-251789322-5050-5407-O5
Task 4 is in state TASK_RUNNING
Task 4 is in state TASK_FINISHED
I1228 08:53:15.337805 6059 sched.cpp:1286] Asked to stop the driver
I1228 08:53:15.338147 6059 sched.cpp:752] Stopping framework
'20141228-085231-251789322-5050-5407-0001'
I1228 08:53:15.338543 6044 sched.cpp:1286] Asked to stop the driver
```

如上输出显示框架连接到 master，并从 slave 接收了资源 offer。同时也显示了所启动任务的各种状态。Java 示例框架在目录 `src/example/java` 下。

```
ubuntu@master:~/mesos/build/src/examples/java $ ./test-framework
master:5050
I1228 08:54:39.290570 7224 sched.cpp:137] Version: 0.21.0
I1228 08:54:39.302083 7250 sched.cpp:234] New master detected at master@
master:5050
I1228 08:54:39.302613 7250 sched.cpp:242] No credentials provided.
Attempting to register without authentication
I1228 08:54:39.307786 7250 sched.cpp:408] Framework registered with
20141228-085231-251789322-5050-5407-0002
Registered! ID = 20141228-085231-251789322-5050-5407-0002
Received offer 20141228-085231-251789322-5050-5407-O6 with cpus: 2.0 and
mem: 1961.0
Launching task 0 using offer 20141228-085231-251789322-5050-5407-O6
Launching task 1 using offer 20141228-085231-251789322-5050-5407-O6
Status update: task 1 is in state TASK_RUNNING
Status update: task 0 is in state TASK_RUNNING
Status update: task 1 is in state TASK_FINISHED
Finished tasks: 1
```

```
Status update: task 0 is in state TASK_FINISHED
Finished tasks: 2
Received offer 20141228-085231-251789322-5050-5407-O7 with cpus: 2.0 and
mem: 1961.0
Launching task 2 using offer 20141228-085231-251789322-5050-5407-O7
Launching task 3 using offer 20141228-085231-251789322-5050-5407-O7
Status update: task 2 is in state TASK_RUNNING
Status update: task 2 is in state TASK_FINISHED
Finished tasks: 3
Status update: task 3 is in state TASK_RUNNING
Status update: task 3 is in state TASK_FINISHED
Finished tasks: 4
Received offer 20141228-085231-251789322-5050-5407-O8 with cpus: 2.0 and
mem: 1961.0
Launching task 4 using offer 20141228-085231-251789322-5050-5407-O8
Status update: task 4 is in state TASK_RUNNING
Status update: task 4 is in state TASK_FINISHED
Finished tasks: 5
I1228 08:54:41.788455 7248 sched.cpp:1286] Asked to stop the driver
I1228 08:54:41.788652 7248 sched.cpp:752] Stopping framework
'20141228-085231-251789322-5050-5407-0002'
I1228 08:54:41.789008 7224 sched.cpp:1286] Asked to stop the driver
```

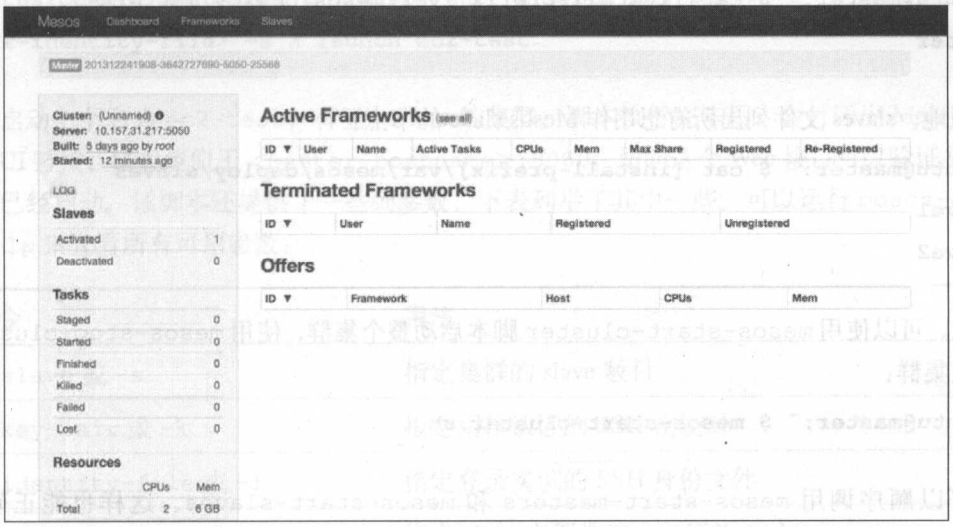
类似地，Python 示例框架在目录 `src/example/python` 下，输出还会显示 `frameworkId` 和各种任务状态：

```
ubuntu@master:~/mesos/build/src/examples/python $./test-framework master:
5050
I1228 08:55:52.389428 8516 sched.cpp:137] Version: 0.21.0
I1228 08:55:52.422859 8562 sched.cpp:234] New master detected at master@
master:5050
I1228 08:55:52.424178 8562 sched.cpp:242] No credentials provided.
Attempting to register without authentication
I1228 08:55:52.428395 8562 sched.cpp:408] Framework registered with
20141228-085231-251789322-5050-5407-0003
Registered with framework ID 20141228-085231-251789322-5050-5407-0003
```

```
Received offer 20141228-085231-251789322-5050-5407-09 with cpus: 2.0 and
mem: 1961.0
Launching task 0 using offer 20141228-085231-251789322-5050-5407-09
Launching task 1 using offer 20141228-085231-251789322-5050-5407-09
Task 0 is in state TASK_RUNNING
Task 1 is in state TASK_RUNNING
Task 0 is in state TASK_FINISHED
Received message: 'data with a \x00 byte'
Task 1 is in state TASK_FINISHED
Received message: 'data with a \x00 byte'
Received offer 20141228-085231-251789322-5050-5407-010 with cpus: 2.0 and
mem: 1961.0
Launching task 2 using offer 20141228-085231-251789322-5050-5407-010
Launching task 3 using offer 20141228-085231-251789322-5050-5407-010
Task 2 is in state TASK_RUNNING
Task 2 is in state TASK_FINISHED
Task 3 is in state TASK_RUNNING
Task 3 is in state TASK_FINISHED
Received message: 'data with a \x00 byte'
Received message: 'data with a \x00 byte'
Received offer 20141228-085231-251789322-5050-5407-011 with cpus: 2.0 and
mem: 1961.0
Launching task 4 using offer 20141228-085231-251789322-5050-5407-011
Task 4 is in state TASK_RUNNING
Task 4 is in state TASK_FINISHED
All tasks done, waiting for final framework message
Received message: 'data with a \x00 byte'
All tasks done, and all messages received, exiting
I1228 08:55:54.136085 8561 sched.cpp:1286] Asked to stop the driver
I1228 08:55:54.136147 8561 sched.cpp:752] Stopping framework
'20141228-085231-251789322-5050-5407-0003'
I1228 08:55:54.136261 8516 sched.cpp:1286] Asked to stop the driver
```

Mesos Web UI

Mesos 提供 Web UI 来展示 Mesos 集群信息。可以从 `<master-host>:<port>` 处访问。示例中是 `http://master:5050`。报告里包括 slave、聚合资源、框架等信息。如下是 Web 接口的截图：



Mesos Web 接口

多节点 Mesos 集群

手动重复之前介绍的步骤，在每个 slave 节点上启动 `mesos-slave`，从而启动整个集群，但是在大规模集群上，这样的手动操作既费时又容易出错。Mesos 的 deploy 目录包含一系列脚本，可以用来在集群上部署 Mesos。这些脚本依赖于 SSH 完成部署。需要搭建免密码登录的 SSH。下面开始搭建集群，包括两个 slave 节点（`slave1`、`slave2`）和一个 master 节点（`master`）。

在所有节点上都安装好所需依赖软件包之后，完成集群配置，确保节点机器之间可以互相访问。如下命令会生成 SSH 密钥，并将其复制到 slave 上：

```
ubuntu@master:~ $ ssh-keygen -f ~/.ssh/id_rsa -P ""
ubuntu@master:~ $ ssh-copy-id -i ~/.ssh/id_rsa.pub ubuntu@slave1
ubuntu@master:~ $ ssh-copy-id -i ~/.ssh/id_rsa.pub ubuntu@slave2
```

将编译后的 Mesos 复制到两个 slave 上，放到和 master 相同的目录下：

```
ubuntu@master:~ $ scp -R build slave1:[install-prefix]
ubuntu@master:~ $ scp -R build slave2:[install-prefix]
```

在 [install-prefix]/var/mesos/deploy/masters 目录创建 masters 文件，可以使用任意编辑器，每一行表示一个 master，在本节示例中，只有一行，如下：

```
ubuntu@master:~ $ cat [install-prefix]/var/mesos/deploy/masters
master
```

类似地，slaves 文件列出所有想用作 Mesos slave 的节点：

```
ubuntu@master:~ $ cat [install-prefix]/var/mesos/deploy/slaves
slave1
slave2
```

现在，可以使用 `mesos-start-cluster` 脚本启动整个集群，使用 `mesos-stop-cluster` 停止集群：

```
ubuntu@master:~ $ mesos-start-cluster.sh
```

也可以顺序调用 `mesos-start-masters` 和 `mesos-start-slaves`，这样也能正确地在 master 节点和 slave 节点上启动相应程序。该脚本会读取 `[install-prefix]/var/mesos/deploy/mesos-deploy-env.sh` 里的环境配置。另外，为了更好地进行配置管理，master 和 slave 的配置参数可以分别定义到独立的 `[install-prefix]/var/mesos/deploy/mesos-master-env.sh` 和 `[install-prefix]/var/mesos/deploy/mesos-slave-env.sh` 文件中。

Amazon EC2 上的 Mesos 集群

Amazon 的 **Elastic Compute Cloud**（**EC2**）借助于虚拟机技术，提供了按使用情况付费的资源访问计算能力，是试验 Mesos 的绝佳方式。Mesos 提供脚本辅助实现在 EC2 上创建各种不同配置的 Mesos 集群。`mesos-ec2` 脚本位于 `ec2` 目录下，允许启动、运行作业，卸载整个 Mesos 集群。注意，即使没有构建 Mesos，我们也可以使用该脚本，但是需要安装版本 >=2.6 的 Python。可以使用不同名称管理多个集群。

要使用 `ec2` 脚本，需要 AWS 密钥对、access key 和 access secret。通过环境变量定义 access key 和 access secret。使用 AWS 管理控制台（`https://console.aws.amazon.com/console/home`）创建并下载密钥对，并赋予 600 权限：

```
ubuntu@local:~ $ chmod 600 my-aws-key.pem
ubuntu@local:~ $ export AWS_ACCESS_KEY_ID=<your-access-key>
ubuntu@local:~ $ export AWS_SECRET_ACCESS_KEY=<your-secret-key>
```

现在就可以使用 Mesos 提供的 EC2 脚本来启动新集群了，使用如下命令：

```
ubuntu@local:~/mesos/ec2 $ ./mesos-ec2 -k <your-key-pair> -i
<your-identity-file> -s 3 launch ec2-test
```

这会启动一个名为 ec2-test，有三个 slave 的集群。脚本执行完毕后，会打印出 Mesos 的 Web UI 链接，格式类似于 <master-hostname>:8080。访问这个 Web 接口可以验证集群是否已经启动。该脚本还提供了一些列参数，下表列举了其中一些。可以运行 mesos-ec2 --help 来查看所有可用参数：

命令	用法
--slave 或 -s	指定集群的 slave 数目
--key-pair 或 -k	指定用作认证的 SSH 密钥对
--identity-file 或 -i	指定登录实例的 SSH 身份文件
--instance-type 或 -t	指定 slave 实例类型，必须为 64-bit
--ebs-vol-size	指定用来存储持久化 HDFS 数据的 EBS 卷的大小
--master-instance-type 或 -m	指定 master 实例类型，必须为 64-bit
--zone 或 -z	指定能够启动实例的 Amazon 可用 zone
--resume	该参数从上一次运行中恢复安装

可以使用 login 操作，通过集群名称登录到已启动的集群上，命令如下：

```
ubuntu@local:~/mesos/ec2 $ ./mesos-ec2 -k <your-key-pair> -i
<your-identity-file> login ec2-test
```

该脚本还会在 /root/ephemeral-hdfs/ 目录下搭建 HDFS 实例，可以通过命令使用该实例。

最后，如下命令可以终止集群。在终止某个集群前一定要备份好重要数据：

```
ubuntu@local:~/mesos/ec2 $ ./mesos-ec2 destroy ec2-test
```

该脚本还支持高级功能，比如用 EBS 后台实例来暂停和重启集群。任何不清楚的地方请参照 Mesos 文档。另外要注意，Mesosphere（http://mesosphere.com）提供了在 Amazon EC2、Google Cloud 及其他平台上创建弹性 Mesos 集群的便捷方式，和对 Mesos 的商业支持。

使用 Vagrant 运行 Mesos

Vagrant 提供了很好的方式来创建便携的虚拟环境，使用它可以很容易地在虚拟机里运行 Mesos。本节讲解如何使用 Vagrant 在虚拟机上创建单节点和多节点 Mesos 集群：

1. 从 https://www.vagrantup.com/download.html 上下载并安装 Vagrant。Vagrant 可以运行在所有主流操作系统上。
2. 使用 Vagrant 插件搭建 Vagrant。执行如下命令完成安装：

```
ubuntu@local:~ $ vagrant plugin install vagrant-omnibus
vagrant-berkshelf vagrant-hosts vagrant-cachier vagrant-aws
```

3. 从 https://github.com/everpeace/vagrant-mesos/ 下载 Vagrant 配置文件，或者使用 git 克隆文件，并 cd 到目录下：

```
ubuntu@local:~ $ git clone https://github.com/everpeace/vagrant-
mesos.git ; cd vagrant-mesos
```

4. 对于单节点集群搭建，cd 到 standalone 目录，运行 vagrant up 命令。这样会创建一台运行 Mesos master、slave 和 ZooKeeper 实例的虚拟机。从 http://192.168.33.10:5050: 可以访问到 Mesos UI：

```
ubuntu@local:~ $ cd standalone ; vagrant up
```

5. 对于多节点集群搭建，cd 到 multinode 目录。在 cluster.yml 文件里配置创建多少台虚拟机用于 Mesos master、slave 和 ZooKeeper 实例。默认会创建五个虚拟机，一台运行 ZooKeeper，两台运行 Mesos master 实例，两台运行 Mesos slave 实例。从 http://172.31.1.11:5050 可以访问多节点环境的 Mesos Web UI：

```
ubuntu@local:~ $ cd multinode ; vagrant up
```

6. Mesos 集群已经启动并运行。可以通过 vagrant ssh 命令登录到这些机器上。单节点集群里使用 master 和 slave 作为主机名，而在多节点集群里，主机名是 master1、slave1 等：

```
ubuntu@local:~ $ vagrant ssh master # to login to master
```

```
ubuntu@local:~ $ vagrant ssh slave # to login to slave
```

7. 可以使用 `halt` 命令停止虚拟机。之后使用 `up` 命令再次启动虚拟机及整个 Mesos 集群。最后，`destroy` 命令会删除所有 Vagrant 创建的虚拟机。注意，必须相应地在 `standalone` 或者 `multinode` 目录下执行 vagrant `destroy` 命令：

```
ubuntu@local:~ $ vagrant halt
ubuntu@local:~ $ vagrant destroy
```

Vagrant 搭建方式也允许很多不同的配置，同时支持启动 Amazon EC2 上的 Mesos 集群。仓库里包含的 vagrant 文件和 `README` 文件里有更多详细内容。

Mesos 社区

虽然还是个较新的项目，Mesos 已经拥有了表现出色的社区（`http://mesos.apache.org/community/`）。小型和大型企业都有很多使用 Mesos 的成功案例（`http://mesos.apache.org/documentation/latest/powered-by-mesos/`）。企业可以使用 Mesos 干很多事情，从网络服务的数据分析到数据存储框架。

案例研究

很多企业将 Mesos 用于生产环境来简化基础架构管理。本节列举了一些案例，介绍这些公司是如何使用 Mesos 的。

Twitter

Twitter 是 Mesos 的第一位使用者，其在 Apache 还处于孵化阶段时就向促进 Mesos 项目发展伸出援手。Twitter 是一个实时社交平台，在可靠的基础架构的帮助下，解决了著名的“失败鲸”难题。Twitter 将 Mesos 作为其整个基础架构的基础，在 Mesos 平台上运行各种各样的作业，包括分析、广告平台、预输入服务和消息基础架构。所有 Twitter 构建的新服务都是 Mesos 支撑的，更为重要的是，Mesos 帮助改变了开发人员对分布式环境资源的理解。现在开发人员会考虑使用资源的共享池，而不是像以前那样只考虑单台机器的资源。Twitter 还构建了 Aurora 调度框架来管理在 Mesos 上长期运行的服务。

HubSpot

HubSpot 制造集客营销的产品。HubSpot 在 Amazon EC2 上运行 Mesos，支持 150 多种不同类型的服务。Mesos 改进了资源使用率，在无须运行服务的多个拷贝的前提下确保了高可用性，降低了基础架构的花费。HubSpot 意识到使用 Mesos 开发人员能够更快地启动新服务，服务能够更可靠更容易扩展。HubSpot 在 Mesos 上创建了 Singularity 框架，并且构建了平台即服务（Platform-as-a-Servie，PaaS）来促进服务的标准化部署。

Airbnb

Airbnb 是社区驱动的租赁公司，也是 Mesos 的最早使用者之一。Airbnb 在 Mesos 上使用 Hadoop、Spark、Kafka 及 Cassandra 和 Rails 这样的服务来完成数据分析。Airbnb 也为 Mesos 创建了 Chronos 调度框架，本书会在第 5 章里详细讨论 Aurora 和 Chronos。

Twitter 的代码构建在 Ruby on Rails 和 JBoss-esque 框架上，天生就是基于服务的，而 Airbnb，从另一方面，更多地使用 Mesos 进行数据处理，天然是 ETL。Twitter 在私有基础架构上，使用 Solaris Zone 将 Mesos 运行在物理硬件上。而 Airbnb 使用 AWS 上基于 VMware 和 Xen hypervisor 的虚拟机。这些案例都有力地证明了 Mesos 提供了通用且易于使用的 API，能够胜任现代分布式基础框架的内核这一角色，可以在多种不同的硬件上运行，并在其上为多种框架提供服务。

邮件列表

在 `http://mesos.apache.org/documentation/latest/` 上有很多与 Mesos 相关的在线文档，详细讲解了 Mesos 的绝大部分内容。当文档信息不够时，Mesos 的用户邮件列表是很好的平台，它使用户可以和其他成员交互，也是 Mesos 社区的关键组成部分。用户邮件列表（`user@mesos.apache.org`）和开发者邮件列表（`dev@mesos.apache.org`）都非常活跃，可以讨论 Mesos 的开发和使用问题。

小结

本章首先讲解了现代集群管理框架的概述，随后又演示了如何搭建 Mesos 集群。至此，已经可以在 Mesos 上运行多种框架了，接下来的几章会对它们进行深入介绍。下一章将介绍 Mesos 上的 Hadoop 框架。

第2章

在 Mesos 上运行 Hadoop

Apache Hadoop 是业界影响力最大的分布式数据处理框架。Hadoop 示范了如何在普通硬件上完成大规模的数据处理。本章详解在 Mesos 上运行 Hadoop 集群的步骤，包括如下内容：

- Hadoop 介绍
- Mesos 上的 Hadoop
- 在 Mesos 上安装 Hadoop
- Hadoop 作业示例
- Mesos 上 Hadoop 的高级配置

Hadoop 介绍

Apache Hadoop（`https://hadoop.apache.org`）诞生于大规模搜索引擎 Nutch。Hadoop 是 MapReduce 范例的实现，Google 著名的 MapReduce 论文发表之后，MapReduce 盛行一时。该项目的成功证明了 MapReduce 模型计算适用于很多真实的业务场景。

Hadoop 项目主要包括两部分：Hadoop MapReduce 和 Hadoop 分布式文件系统（Hadoop Distributed File System，HDFS）。HDFS 可以在普通硬件上构建可扩展、可容错的分布式文件系统。HDFS 包括一个或多个 Namenode 及多个 Datanode。Hadoop Namenode 是分布式文件系统（Distributed File System，DFS）的 master，负责存储文件系统的所有元数据。Hadoop Datanode 是 DFS 的 slave，存储实际数据。Hadoop 依靠复制实现容错和吞吐量。也可以单独使用 HDFS，很多公司将其作为 Mesos 的分布式文件系统。

Hadoop MapReduce 是分布式执行引擎。Hadoop 作业来自于用户程序，并被分解成多个任务。这些任务可能是 Map 或者 Reduce 任务，负责处理一份数据。Hadoop MapReduce 包括 JobTracker 和多个 TaskTracker。JobTracker 是在 Hadoop 框架里负责协调作业的组件。它作为主程序，在 TaskTracker 上调度任务。Hadoop TaskTracker 则是 slave 程序，为应用程序执行任务。

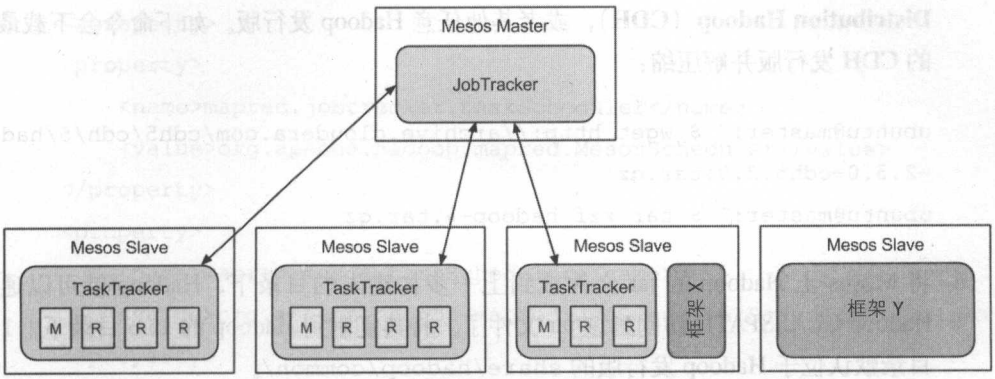
本节简要介绍 Hadoop。要想获得 Hadoop 更多更详细的信息，可以参考 Tom White 撰写的权威指南或 Hadoop 官方文档（`http://hadoop.apache.org`）。

最近，Hadoop MapReduce 项目分成了两个主要组件：Hadoop YARN 和 Hadoop MapReduce，YARN 处理多种框架，包括 MapReduce 的资源分配。Mesos 上的 Hadoop 目前不支持 MapReduce version 2（MRv2）或 YARN。传统的 Hadoop 称为 MapReduce1 或 MRv1。Apache 孵化项目 Myriad（`https://github.com/mesos/myriad`）帮助你在 Mesos 上运行弹性 YARN 集群。在 Mesos 上运行 YARN，可以实现 YARN 和 Mesos 上应用程序间的资源共享。Myriad 项目是由 eBay 发起的，本书撰写时，它已经被广泛部署。活跃的社区力量也推动 standalone HDFS 成为 Mesos 框架（`https://github.com/mesosphere/hdfs`）之一。这使得 HDFS 可以应对故障，在和其他应用程序共享资源时，HDFS 非常有用，它也为简化运维方式提供了帮助。

Mesos 上的 Hadoop

在 Mesos 上运行 Hadoop 可以实现多框架共享集群资源。通过在 Mesos 上运行 Hadoop，我们可以借助广大社区的力量，使用所有构建在 Hadoop 之上的框架，比如，Giraph、Hama、HBase、Hive，等等。另外，如果已经使用了 Hadoop，将其运行到 Mesos 上，可以继续使用围绕 Hadoop 生态系统构建的工具，同时又能提高资源使用率。现有的 MapReduce 代码和工具都可以在 Mesos 上的 Hadoop 上使用。使用 Mesos 上的 Hadoop，还可以在同一个 Mesos 集群资源上运行多版本的 Hadoop。Mesos 上的 Hadoop 项目围绕 Hadoop 的 JobTracker 实现了 Mesos 框架的调度器，并且将 Hadoop 执行器封装成 Mesos 执行器。下图展现了 Hadoop 和 Mesos 的集成方式。

下图中，Mesos master 运行 JobTracker，一些 Mesos slave 运行 TaskTracker。每个 TaskTracker 都运行着一些 Map 和 Reduce 的任务。注意 Mesos 上的 Hadoop 和其他框架（图中的 X 和 Y）共享资源。



Mesos 上运行的 Hadoop MapReduce 进程

在 Mesos 上安装 Hadoop

Mesos 上的 Hadoop（https://github.com/mesos/hadoop）依赖于 Mesos 的扩展，比如，Mesos 执行器来执行 TaskTracker，以及 Hadoop JobTracker Mesos 调度器在 Mesos 框架上运行 Hadoop。本节介绍如何在 Mesos 上运行 Hadoop 1.x：

1. 按照第 1 章的说明，安装并运行 Mesos。
2. 需要基于 Mesos 库来编译 Hadoop。Mesos 上的 Hadoop 使用 Maven 来管理依赖关系，使用如下命令安装 Maven、Java 和 Git。

```
ubuntu@master:~ $ sudo apt-get install maven openjdk-7-jdk git
```

3. 从 https://github.com/mesos/hadoop 处克隆 Mesos 上 Hadoop 的源代码，cd 到根目录：

```
ubuntu@master:~ $ git clone https://github.com/mesos/hadoop/
ubuntu@master:~ $ cd hadoop
```

4. 使用如下命令，从源码构建出 Mesos 上 Hadoop 的可执行二进制文件。默认会构建最新版本的 Mesos 和 Hadoop。如果需要，可以在 pom.xml 文件里指定所需版本。

```
ubuntu@master:~ $ mvn package
```

命令执行完，会在 target 目录下构建出 hadoop-mesos-VERSION-jar。

5. 下载 Hadoop，解压缩，进入根目录。可以使用 vanilla Apache 发行版、**Cloudera Distribution Hadoop（CDH）**，或者其他任意 Hadoop 发行版。如下命令会下载最新的 CDH 发行版并解压缩：

```
ubuntu@master:~ $ wget http://archive.cloudera.com/cdh5/cdh/5/hadoop
-2.5.0-cdh5.2.0.tar.gz
ubuntu@master:~ $ tar xzf hadoop-*.tar.gz
```

6. 将 Mesos 上 Hadoop 的 jar 文件放到上一步构建出的目录下，Hadoop 就可以通过 Hadoop CLASSPATH 访问到该 jar 文件了。将其复制到 Hadoop 的 `lib` 目录下，`lib` 目录默认位于 Hadoop 发行版的 `share/hadoop/common/`。

```
ubuntu@master:~ $ cp hadoop-mesos/target/hadoop-mesos-*.jar
hadoop-*/share/hadoop/common/lib
```

7. CDH 发行版默认使用带有 YARN 的 MapReduce version 2（MRv2）。因此，需要更新配置指向 MRv1：

```
ubuntu@master:~ $ cd hadoop-*

ubuntu@master:~ $ mv bin bin-mapreduce2
ubuntu@master:~ $ ln -s bin-mapreduce1 bin

ubuntu@master:~ $ cd etc;
ubuntu@master:~ $ mv hadoop hadoop-mapreduce2
ubuntu@master:~ $ ln -s hadoop-mapreduce1 hadoop
ubuntu@master:~ $ cd -;
```

另外，可以选择更新样例，指向 MRV1 的样例：

```
ubuntu@master:~ $ mv examples examples-mapreduce2
ubuntu@master:~ $ ln -s example-mapreduce1 examples
```

8. 现在需要配置 Hadoop 来使用刚刚构建的 Mesos 调度器。在 `etc/hadoop/mapred-site.xml` 里设置如下必选配置参数，需要将参数添加到 `<configuration>` 和 `</configuration>` 标签之间：

```
<property>
    <name>mapred.job.tracker</name>
```

```
    <value>localhost:9001</value>
</property>
<property>
    <name>mapred.jobtracker.taskScheduler</name>
    <value>org.apache.hadoop.mapred.MesosScheduler</value>
</property>
<property>
    <name>mapred.mesos.taskScheduler</name>
    <value>org.apache.hadoop.mapred.JobQueueTaskScheduler</value>
</property>
<property>
    <name>mapred.mesos.master</name>
    <value>zk://localhost:2181/mesos</value>
</property>
<property>
    <name>mapred.mesos.executor.uri</name>
    <value>hdfs://localhost:9000/hadoop.tar.gz</value>
</property>
```

通过指定 `mapred.jobtracker.taskScheduler` 属性指定 Hadoop 使用 Mesos 来调度任务。通过 `mapred.mesos.master` 指定 Mesos master 地址，本例将其设为本地 ZooKeeper 的地址。`mapred.mesos.executor.uri` 指向会上传到 HDFS 的 Hadoop 发行版路径，用来完成任务的执行。

9. 必须确保 Mesos 上的 Hadoop 能够找到 Mesos 原生库，该库默认位于 `/usr/local/lib/libmesos.so`。通过添加如下行到 `/bin/hadoop-daemon.sh` 脚本的开头来 export 出 Mesos 原生库的路径。

```
export MESOS_NATIVE_LIBRARY=/usr/local/lib/libmesos.so
```

10. 需要设定一个路径，让 Mesos 可以在启动 Hadoop 任务时通过该路径访问到 Hadoop 的发行版。可以将该路径放在 HDFS、S3 或任意其他可访问的位置，比如 NFS 服务器。本例将其放在 HDFS 上，为此需要安装 HDFS master 节点。注意，HDFS master 节点独立于 Mesos master。将 Hadoop 发行版复制到该节点，使用如下命令启动 Namenode：

```
ubuntu@master:~$ bin/hadoop-daemon.sh start namenode
```

需要在作为 HDFS slave 节点（独立于 Mesos slave）的每台节点机器上启动 Datanode

守护进程。将 Hadoop 发行版复制到所有 HDFS slave 节点上，然后使用如下命令启动 Datanode：

```
ubuntu@master:~$ bin/hadoop-daemon.sh start datanode
```

在 Namenode 运行着的 HDFS master 上，运行如下命令完成第一次使用时 Namenode 的格式化：

```
ubuntu@master:~$ bin/hadoop namenode -format
```

11. Hadoop 已经可以在 Mesos 上运行了。打包并将其上传到 HDFS 上：

```
ubuntu@master:~ $ tar cfz hadoop.tar.gz hadoop-*
ubuntu@master:~ $ bin/hadoop dfs -put hadoop.tar.gz /hadoop.tar.gz
ubuntu@master:~ $ bin/hadoop dfs -chmod 777 /hadoop.tar.gz
```

12. 现在就可以启动 JobTracker 了。注意，不需要手动启动 TaskTracker，因为在提交 Hadoop 作业时，Mesos 会自动启动 TaskTracker。

```
ubuntu@master:~ $ bin/hadoop jobtracker
```

Hadoop 开始运行，一切就绪，可以运行 Hadoop 作业了。

Hadoop 作业示例

运行一个示例作业，该作业计录输入文件中每个单词的词频。

1. 需要一些文本作为输入来运行该示例。从 Project Gutenberg 的网站上下载《爱丽丝奇遇记》：

```
ubuntu@master:~ $ wget http://www.gutenberg.org/files/11/11.txt -O
/tmp/alice.txt
```

2. 在 HDFS 上创建 input 目录，将下载好的文件放入该目录：

```
ubuntu@master:~ $ bin/hadoop dfs -mkdir input
ubuntu@master:~ $ bin/hadoop dfs -put /temp/alice.txt input
```

3. 从 Hadoop 发行版自带的 hadoop-examples.jar 文件里运行 Hadoop WordCount 示例：

```
ubuntu@master:~ $ bin/hadoop jar hadoop-examples.jar wordcount input
output
```

4. 程序会输出到 HDFS 的 output 目录。可使用如下命令查看输出：

```
ubuntu@master:~ $ bin/hadoop dfs -cat output/*
```

输出会在每一行列出输入文本中出现的单词及其词频。

Mesos 上 Hadoop 的高级配置

Hadoop 有很多参数会影响到其性能和容错行为。需要根据集群和工作负载的特点对如下参数进行调优。

任务资源分配

该参数用来控制分配给 Hadoop 任务的资源总数。在 conf/core-site.xml 上设置如下参数，会给 Mesos 分配合适的 CPU、内存（MB 为单位）及磁盘空间（MB 为单位）资源。

注意分配的内存包括 JVM 额外开销（~10%）。

```
<property>
    <name>mapred.mesos.slot.cpus</name>
    <value>1</value>
</property>
<property>
    <name>mapred.mesos.slot.mem</name>
    <value>1024</value>
</property>
<property>
    <name>mapred.mesos.slot.disk</name>
    <value>1024</value>
</property>
```

如下参数控制 Mesos 的资源策略：

- mapred.mesos.total.map.slots.minimum 和 mapred.mesos.total.reduce.slots.minimum 指定在限定时间内 Mesos 尝试获得 Map 和 Reduce slot 的最小数量。这个数目仅作参考，不保证这些 slot 都是可用的。类似的，mapred.tasktracker.map.tasks.maximum 和 mapred.tasktracker.reduce.tasks.maximum 控制此类 slot 的最大数目：

```
<property>
    <name>mapred.mesos.total.map.slots.minimum</name>
    <value>0</value>
</property>
<property>
    <name>mapred.mesos.total.reduce.slots.minimum</name>
    <value>0</value>
</property>
<property>
    <name>mapred.tasktracker.map.tasks.maximum</name>
    <value>50</value>
</property>
<property>
    <name>mapred.tasktracker.reduce.tasks.maximum</name>
    <value>50</value>
</property>
```

- 如果想要 Mesos 总是基于最大 Map/Reduce slot 参数为每个 TaskTracker 分配固定数量的 slot 并拒接其他 offer，可以将 mapred.mesos.scheduler.policy.fixed 设置为 true：

```
<property>
    <name>mapred.mesos.scheduler.policy.fixed</name>
    <value>false</value>
</property>
```

有更早的版本通过监控空闲 TaskTracker 来更为弹性地扩展 TaskTracker。这样的版本里，空闲的 TaskTracker 会被杀死，其资源就可以被 Mesos 集群上的其他框架使用。有两个配置参数控制该行为：

`mapred.mesos.tracker.idle.interval` 控制检查 TaskTracker 使用率的频率。间隔以秒计，默认是 5 秒。

`mapred.mesos.tracker.idle.checks` 指定在杀死 TaskTracker 之前需要成功认定其为空闲的次数。默认是 5 次，也就是说，5 次成功检测到 TaskTracker 为空闲时，该 TaskTracker 就会被杀死。

度量报告

对于任何大规模集群而言，监控都极其重要。Mesos 支持从 Hadoop 导出适合外部系统的度量报告。使用 Coda Hale Metrics 库（`http://metrics.codahale.com/`）来实现度量。要启用度量，需要在 `conf/core-site.xml` 里做如下设置。该功能默认是关闭的：

```
<property>
    <name>mapred.mesos.metrics.enabled</name>
    <value>true</value>
</property>
```

度量报告可以导出到多种系统里。如下几种是使用最广泛的。

CSV

启用 **Comma Separated Values**（**CSV**）报告，使用正确的路径设置如下配置。会在 `mapred.mesos.metrics.csv.path` 指定的路径下创建 CSV 文件，来报告度量结果，同时 `mapred.mesos.metrics.csv.interval` 控制生成报告的频率：

```
<property>
    <name>mapred.mesos.metrics.csv.enabled</name>
    <value>true</value>
</property>
<property>
    <name>mapred.mesos.metrics.csv.path</name>
    <value>/path/to/metrics/csv/metrics.csv</value>
</property>
<property>
```

```
    <name>mapred.mesos.metrics.csv.interval</name>
    <value>60</value>
</property>
```

Graphite

Graphite（http://graphite.wikidot.com）是非常流行的开源工具，用来监控及图形化性能数据。Graphite 是监控集群的最佳实时图形化框架之一：

```
<property>
    <name>mapred.mesos.metrics.graphite.enabled</name>
    <value>true</value>
</property>
<property>
    <name>mapred.mesos.metrics.graphite.host</name>
    <value>your_graphite_host_name</value>
</property>
<property>
    <name>mapred.mesos.metrics.graphite.port</name>
    <value>your_graphite_port</value>
</property>
<property>
    <name>mapred.mesos.metrics.graphite.prefix</name>
    <value>your_graphite_prefix</value>
</property>
<property>
    <name>mapred.mesos.metrics.graphite.interval</name>
    <value>60</value>
</property>
```

Cassandra

如果有其他服务系统也需要度量，或者想要将度量数据存储下来以供后续分析，可以选择 Cassandra：

```
<property>
    <name>mapred.mesos.metrics.cassandra.enabled</name>
```

```
    <value>true</value>
</property>
<property>
    <name>mapred.mesos.metrics.cassandra.hosts</name>
    <value>your_cassandra_host</value>
</property>
<property>
    <name>mapred.mesos.metrics.cassandra.port</name>
    <value>your_cassandra_port(normally 9042)</value>
</property>
<property>
    <name>mapred.mesos.metrics.cassandra.interval</name>
    <value>60</value>
</property>
<property>
    <name>mapred.mesos.metrics.cassandra.prefix</name>
    <value>your_cassandra_prefix</value>
</property>
<property>
    <name>mapred.mesos.metrics.cassandra.ttl</name>
    <value>864000</value>
</property>
<property>
    <name>mapred.mesos.metrics.cassandra.keyspace</name>
    <value>metrics</value>
</property>
<property>
    <name>mapred.mesos.metrics.cassandra.table</name>
    <value>metrics</value>
</property>
<property>
    <name>mapred.mesos.metrics.cassandra.consistency</name>
    <value>QUORUM</value>
</property>
```

认证

Mesos 上的 Hadoop 支持如下配置参数来完成所需认证。

要想为不同的框架得到 Mesos 的预留权限，需要使用角色（role）。设置 `mapred.mesos.role` 为所需角色。默认值是 `*`，意味着所有框架享有同等的资源权限：

```
<property>
    <name>mapred.mesos.role</name>
    <value>*</value>
</property>
```

如下参数控制 Mesos 上的 Hadoop 是否从其他角色处接受资源 offer。如果设置为 true，框架就只会接受 `mapred.mesos.role` 所配置的角色的资源 offer：

```
<property>
    <name>mapred.mesos.role.strict</name>
    <value>false</value>
</property>
```

`mapred.mesos.framework.principal` 指定 Mesos 框架用于认证的原则：

```
<property>
    <name>mapred.mesos.framework.principal</name>
    <value>hadoop</value>
</property>
```

`mapred.mesos.framework.secretfile` 指定保存 Mesos 框架 secret 文件的路径，该文件也会作为认证的一部分：

```
<property>
    <name>mapred.mesos.framework.secretfile</name>
    <value>/location/secretfile</value>
</property>
```

`mapred.mesos.framework.user` 指定框架运行的用户。默认值设为运行框架调度器的用户：

```
<property>
    <name>mapred.mesos.framework.user</name>
    <value>hadoop</value>
</property>
```

注意，后续章节我们会详细讨论这些参数和术语，现在只是简单地总体介绍。

容器隔离

Mesos 上 Hadoop 的 JobTracker 能够通过 JobConf 发送自定义的容器镜像，该镜像用来在 Mesos 上隔离任务。该功能很有用，可以将 Hadoop 软件依赖程序和 slave Mesos 的其他部分隔离开来。该功能在 Mesos version 0.19 后可用，是可选的，无论是否选择使用容器隔离，Mesos 上的 Hadoop 都能够正常工作。也就是说，如果不设置如下参数，JobConf（来自 Hadoop）就不会向 Mesos 传递任何 `ContainerInfo`。

Mesos 上的 Hadoop 支持通过如下两个配置参数来控制容器。第一个参数指定需要使用的容器镜像，第二个参数指定运行该镜像所需的参数：

```
<property>
    <name>mapred.mesos.container.image</name>
    <value>docker:///ubuntu</value>
</property>
<property>
    <name>mapred.mesos.container.options</name>
    <value>-v,/foo/bar:/bar</value>
</property>
```

使用的容器镜像必须事先安装 Mesos 原生库。

其他配置参数

Hadoop 可以借助于 Mesos 的高可用性，通过启动检查点，在重启后恢复运行。下一章详细讨论 slave 的恢复。要在 Mesos 上的 Hadoop 中开启该功能，需要将 `mapred.mesos.checkpoint` 设置为 `true`：

```
<property>
    <name>mapred.mesos.checkpoint</name>
    <value>true</value>
</property>
```

`mapred.mesos.framework.name` 指定框架名称。默认值为 `hadoop`：

```
<property>
    <name>mapred.mesos.framework.name</name>
```

```
    <value>hadoop</value>
</property>
```

同时，也可以优化在 Mesos 上运行的 Hadoop。比如，将多个任务请求发送给一个 TaskTracker，而不是为每个执行器都启动一个 TaskTracker。社区里有很多关于这些改动的讨论，但是在本书撰写时还没有全部完成，应该很快就可以面世了。

小结

本章介绍了 Hadoop 概述，用示例演示了如何在 Mesos 上运行 Hadoop 集群。同时讲解了一些重要的配置参数，可以用来调优性能和改进运维体验。

下一章会介绍在 Mesos 上如何使用 Spark 进行数据分析。

第3章 在 Mesos 上运行 Spark

Spark 是进行大规模数据处理的快速通用的执行引擎。大规模集群的主要用途之一就是运行数据处理工作。Spark 可以处理多种格式的数据，是 Berkely 数据分析栈（**Berkely Data Analytics Stack，BDAS**）的一部分。Spark 支持批处理、迭代处理、近实时处理和流处理。本章会介绍在 Mesos 集群上搭建 Spark 的详细步骤。本章内容包括：

- Spark 介绍
- Spark 作业调度
- Spark standalone 模式
- 在 Mesos 上的 Spark
- 在 Mesos 上 Spark 的调优

Spark 介绍

Apache Spark 是快速可扩展的通用数据处理框架（`http://spark.apache.org`）。Spark 为编写大规模数据处理应用程序提供了非常简明的语法。它在 2014 年初就成为 Apache 的顶级项目。

Spark 项目和 Mesos 一样，都是 Berkeley 数据分析栈（`https://amplab.cs.berkeley.edu/software`）的一部分，由 Berkeley 启动。Spark 是构建在 Mesos 上的首个数据处理框架，能够有效地使用 Mesos 进行资源管理。Spark 是发展最为迅速的数据分析框架之一，目的是在单套一致标准的 API 下统一所有类型的数据分析。Spark 提供了统一的 API 来完成

批、流和迭代数据处理。

Spark 竭尽全力使用内存来加速计算。它的直接非循环图（Directed Acycylic Graph，DAG）执行引擎适用于各种不同的应用程序，包括交互式和迭代式算法，通常用于机器学习、实时分析和图处理领域。Spark 的引擎十分通用，支持各种高级工具，包括但不限于，用于数据仓库系统的 Spark SQL（https://spark.apache.org/sql），用于流处理应用程序的 Spark Streaming（https://spark.apache.org/streaming），用于可扩展机器应用程序的 MLlib（https://spark.apache.org/mllib），以及用于图处理的 GraphX（https://spark.apache.org/graphx）。Spark 和各种不同的数据源集成。很多公司使用 Mesos 上的 Spark 来洞察数据。

Spark 基于弹性分布式数据集（**Resilient Distributed Datasets，RDD**）的抽象而构建（http://www.cs.berkeley.edu/~matei/papers/2012/nsdi_spark.pdf）。RDD 的研究学者定义 RDD 为只读的分区的记录集合。RDD 是分布式数据集，支持高级的并行操作。它是分布式内存的抽象，允许以分布式的方式进行内存内计算，同时提供容错。RDD 是共享内存的受限形式，允许粗粒度的预定义转化为共享状态。因为不允许细粒度随意更新到共享状态，RDD 在保证容错的同时也不会导致过高的额外消耗。通过限制转化过程，RDD 无须牺牲效率就能够实现容错。另外，传统的分布式共享内存（Distributed Shared Memory，DSM）系统受限于可用内存的总量，但是 RDD 可以无缝地使用磁盘作为内存扩展。因为 RDD 进行批量读/写，而不像 DSM 采取的是随机读/写，所以能够保证平稳的性能。

RDD 转化能够支持各种不同的应用程序，构建于 Spark 之上的各种框架已经证明了它的能力。RDD 通过存储操作链（应用于数据的一系列操作）而不是每次操作后的实际数据来实现容错。当故障发生时，RDD 可以通过应用转化从数据里重新推导出丢失的数据。

注意，操作链可能会特别长，所以可以为 RDD 设置检查点以便在错误发生时快速恢复。设置多少检查点需要在性能和恢复时间之间进行权衡。

Spark 使用 Scala 编写，通过语言绑定，可以支持用 Scala、Java 或 Python 编写的程序。Spark 编程指南详细介绍了 API 和可用转化（https://spark.apache.org/docs/latest/programming-guide.html）。书籍《Spark 快速大数据分析》和《Spark 快速数据处理》有更多深入的内容。

Spark 作业调度

本节介绍在集群里 Spark 作业是如何调度的。Spark 的集群模式涉及作业调度方式和 Spark 应用程序之间的资源管理（https://spark.apache.org/docs/latest/job-

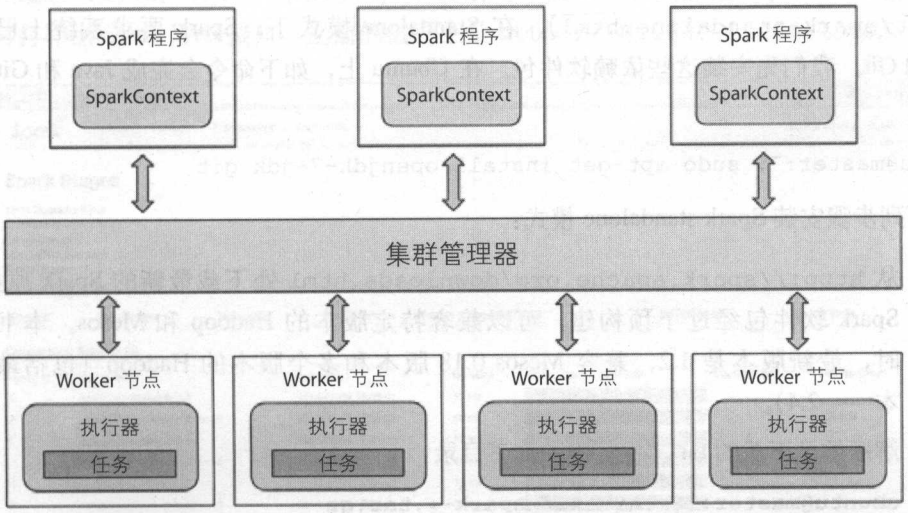
scheduling.html)。Spark 的集群管理器为资源管理提供了两种策略：

- 在静态分区里，每个 Spark 应用程序都会从集群里得到固定总量的资源。应用程序在整个执行期间都占用这些资源。Spark standalone 模式和 Mesos 粗粒度模式都采用这种方式。
- CPU 核的动态共享。该模式下，每个 Spark 应用程序被分配固定总量的内存，但是仅在 Spark 应用运行时 CPU 才会被分配。Mesos 细粒度模式使用这种策略。在有很多空闲 Spark 会话时，这种策略很有用。

这两种模式的内存共享都是固定的，只是在 CPU 共享方式上有所不同。

从 1.2 版本开始，Spark 引入了资源动态扩展的理念。本书撰写时，动态资源扩展只在 Spark Standalone 模式和 YARN 上可用。动态资源扩展的能力使得资源的使用更具弹性。Spark 将来的版本也会在 Mesos 上引入动态资源扩展功能。

下图描述了 Spark 的各种组件：



Spark 架构

如下是 Spark 集群的重要组成部分，https://spark.apache.org/docs/latest/cluster-overview.html 中有详细介绍：

- 应用程序（Application）是构建并运行在 Spark 上的用户程序。应用程序包括驱动程

序和执行器。驱动器运行 `main()` 函数并创建 `SparkContext` 对象。`SparkContext` 管理 Spark 应用程序的执行。

- 作业（Job）指的是Spark操作触发的并行计算。它包含多个称为任务（task）的独立单元。
- 集群管理器（Cluster manager）是负责集群上资源协调的模块。目前 Spark 支持 standalone 集群管理器、YARN 和 Mesos 作为其集群管理器。
- worker 节点是拥有资源的节点，能够在集群里运行 Spark 的应用程序代码。worker 节点启动 Spark 执行器的进程，从应用程序里运行任务。每个应用程序都有自己的执行器，负责管理任务的执行及任务所需的数据。Spark 以执行器的形式获得工作资源，并将应用程序代码发送给 worker 节点。一切就绪后，`SparkContext` 处理任务，运行执行器。

Spark Standalone模式

Spark standalone 模式使用简单的内置集群管理器。Spark standalone 模式集群管理器也能运行在单台机器上，提供了尝试 Spark 的绝佳方式（`http://spark.apache.org/docs/latest/spark-standalone.html`）。在 Standalone 模式下，Spark 要求系统上已安装 Java 和 Git。我们先安装这些依赖软件包。在 Ubuntu 上，如下命令会完成 Java 和 Git 的安装：

```
ubuntu@master:~$ sudo apt-get install openjdk-7-jdk git
```

根据下列步骤安装 Spark standalone 模式：

1. 从 `http://spark.apache.org/downloads.html` 处下载最新的 Spark 原始码。Spark 软件包经过了预构建，可以兼容特定版本的 Hadoop 和 Mesos。本书撰写时，最新版本是 1.2，兼容 Mesos 0.18 版本和多个版本的 Hadoop（包括最新版本——2.4）。
2. 解压缩已下载的 tar 文件，进入如下目录：

   ```
   ubuntu@master:~$ tar -xzf spark-*.tar.gz
   ubuntu@master:~$ cd spark-*
   ```

3. 现在就可以启动 Spark Scala shell 了。Spark shell 是 Scala shell 的修改版，是为了支持交互式分析和初期探测的。与之类似，可以通过 `./bin/pyspark` 启动 Python Spark shell。Spark 提供了很好用的 shell：

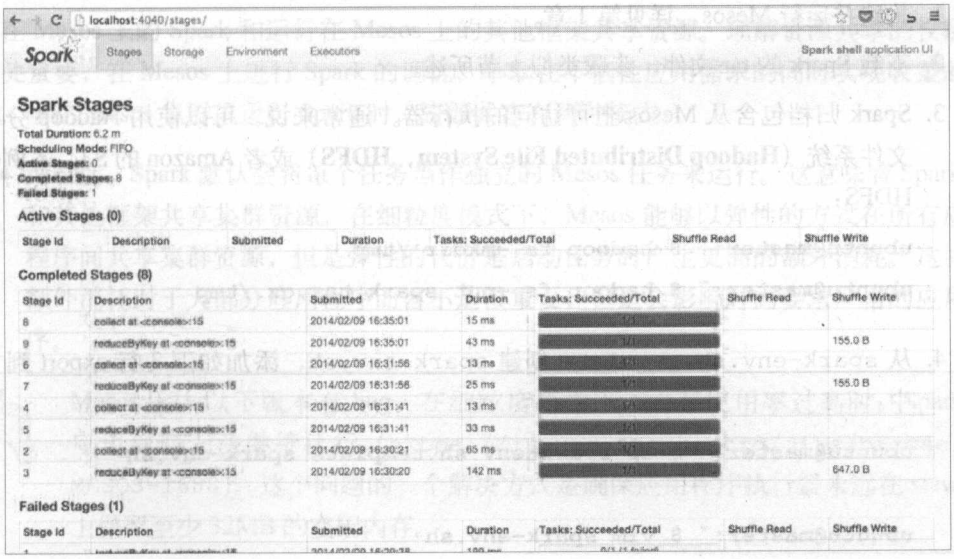
   ```
   ubuntu@master:~ $ ./bin/spark-shell
   ```

```
$ ./spark-shell
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
14/12/26 13:16:01 INFO SecurityManager: Changing view acls to: GreatGod
14/12/26 13:16:01 INFO SecurityManager: Changing modify acls to: GreatGod
14/12/26 13:16:01 INFO SecurityManager: SecurityManager: authentication disabled; ui acls disabled; users with view permissions: Set(GreatGod); users with modify permissions: Set(G
reatGod)
14/12/26 13:16:01 INFO HttpServer: Starting HTTP Server
14/12/26 13:16:01 INFO Utils: Successfully started service 'HTTP class server' on port 63217.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.2.0
      /_/

Using Scala version 2.10.4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_05)
Type in expressions to have them evaluated.
Type :help for more information.
14/12/26 13:16:13 INFO SecurityManager: Changing view acls to: GreatGod
14/12/26 13:16:13 INFO SecurityManager: Changing modify acls to: GreatGod
14/12/26 13:16:13 INFO SecurityManager: SecurityManager: authentication disabled; ui acls disabled; users with view permissions: Set(GreatGod); users with modify permissions: Set(G
reatGod)
14/12/26 13:16:15 INFO Slf4jLogger: Slf4jLogger started
14/12/26 13:16:15 INFO Remoting: Starting remoting
14/12/26 13:16:16 INFO Remoting: Remoting started; listening on addresses :[akka.tcp://sparkDriver@10.168.91.107:63218]
14/12/26 13:16:16 INFO Utils: Successfully started service 'sparkDriver' on port 63218.
14/12/26 13:16:16 INFO SparkEnv: Registering MapOutputTracker
14/12/26 13:16:16 INFO SparkEnv: Registering BlockManagerMaster
14/12/26 13:16:16 INFO DiskBlockManager: Created local directory at /var/folders/vx/rw50qrnn1yd7v7cbnl7fdkn80000gn/T/spark-local-20141226131616-22a2
14/12/26 13:16:16 INFO MemoryStore: MemoryStore started with capacity 265.1 MB
14/12/26 13:16:17 INFO HttpFileServer: HTTP File server directory is /var/folders/vx/rw50qrnn1yd7v7cbnl7fdkn80000gn/T/spark-c54b8d62-3af1-4ac4-b174-f2a12ec00160
14/12/26 13:16:17 INFO HttpServer: Starting HTTP Server
14/12/26 13:16:17 INFO Utils: Successfully started service 'HTTP file server' on port 63219.
14/12/26 13:16:18 INFO Utils: Successfully started service 'SparkUI' on port 4040.
14/12/26 13:16:18 INFO SparkUI: Started SparkUI at http://10.168.91.107:4040
14/12/26 13:16:18 INFO Executor: Using REPL class URI: http://10.168.91.107:63217
14/12/26 13:16:18 INFO AkkaUtils: Connecting to HeartbeatReceiver: akka.tcp://sparkDriver@10.168.91.107:63218/user/HeartbeatReceiver
14/12/26 13:16:18 INFO NettyBlockTransferService: Server created on 63220
14/12/26 13:16:18 INFO BlockManagerMaster: Trying to register BlockManager
14/12/26 13:16:18 INFO BlockManagerMasterActor: Registering block manager localhost:63220 with 265.1 MB RAM, BlockManagerId(<driver>, localhost, 63220)
14/12/26 13:16:18 INFO BlockManagerMaster: Registered BlockManager
14/12/26 13:16:19 INFO SparkILoop: Created spark context..
Spark context available as sc.

scala>
```

Spark 还包含 Web 接口，提供了有关应用程序的状态信息。可以从 `http://<driver-node>:4040` 处访问。端口 `4040` 是默认端口，如果多个应用程序同时运行，会使用之后的连续端口（`4041`、`4042`，等等）。Spark 会话运行在 driver 节点上。Spark shell 会在会话开始时打印出 Spark Web 接口，之前的截图上是 `http://10.168.91.107:4040`。



Spark 在 `examples` 目录下包含很多 Spark 的样例程序。脚本 `bin/run-examples` 可以运行这些样例，帮助验证搭建是否成功。比如，运行 Spark PI 近似值样例：

```
ubuntu@master:~ $./bin/run-example org.apache.spark.examples.SparkPi
```

```
SLF4J: Class path contains multiple SLF4J bindings.
....
14/02/09 16:44:04 INFO SparkContext: Job finished: reduce at SparkPi.
scala:39, took 8.638717 s
Pi is roughly 3.13832
....
14/02/09 16:44:06 INFO MapOutputTrackerMasterActor: MapOutputTrackerActor
stopped!
```

一旦 Spark 安装完成，在 Spark 会话运行时就可以查看 Spark 的 Web UI。另外，Mesos 的 Web 接口也会在活动框架上列出 Spark。可以运行其他 Scala 样例及用 Java 或 Python 编写的样例，这些都默认包含在 Spark 的发行版中。

在 Mesos 上的 Spark

Mesos 可以作为 Spark 的集群管理器。当在 Mesos 上运行 Spark 时，Spark 可以利用 Mesos 的资源管理能力，Spark 任务在使用 Spark 执行器的 Mesos worker 节点上执行。这样就可以在多个 Spark 实例间或其他框架间共享资源了。如下步骤为在 Mesos 上安装 Spark：

1. 构建并运行 Mesos，详见第 1 章。
2. 下载 Spark 的 tar 文件，步骤类似上节所述。
3. Spark 归档包含从 Mesos 上可访问的执行器。通常来说，可以使用 Hadoop 分布式文件系统（**Hadoop Distributed File System**，**HDFS**）或者 Amazon 的 S3。本例使用 HDFS：

   ```
   ubuntu@master:~ $ hadoop fs -mkdir /tmp
   ubuntu@master:~ $ hadoop fs -put spark.tar.gz /tmp
   ```

4. 从 `spark-env.sh.template` 创建 `spark-env.sh`，添加如下 3 行 export 到文件中：

   ```
   ubuntu@master:~ $ cp spark-env.sh.template spark-env.sh

   ubuntu@master:~ $ vim spark-env.sh
   export MESOS_NATIVE_LIBRARY=/usr/local/lib/libmesos.so
   export SPARK_EXECUTOR_URI=hdfs://master/tmp/spark.tar.gz
   export MASTER=mesos://master:5050
   ```

MESOS_NATIVE_LIBRARY 指定 libmesos.so 的位置，这是 slave 节点上的 Mesos 库。该库默认安装在 /usr/local/lib 下。SPARK_EXECUTOR_URI 是 Mesos 中可以使用 Spark 发行版启动工作进程的位置。MASTER 指定 Mesos master 地址。这些是需要设置的最少参数。还可以指定其他参数，比如，Spark master 侦听的地址（SPARK_MASTER_IP 和 SPARK_MASTER_PORT）和 Spark 端口（SPARK_WORKER_PORT），等等。

至此，Spark 配置完成，已经可以在 Mesos 集群上运行了。可以创建 SparkContext，指定 Mesos master URI 和 Spark 执行器路径，如下：

```
val conf = new SparkConf()
 .setMaster("mesos://master:5050")
 .set("spark.executor.uri", "hdfs://master/tmp/spark.tar.gz")
val sc = new SparkContext(conf)
```

现在，如果查看 Mesos 上的活动框架，应该能够找到 Spark。还可以查看活动任务的细节，以及可以便捷访问 stderr 和 stdout 的沙箱链接。

在 Mesos 上 Spark 的调优

运行在 Mesos 上的 Spark 和运行在 Mesos 上的其他框架共享资源。理解资源共享的权衡思想至关重要，在 Mesos 上进行 Spark 的调优，争取在不牺牲应用需求的同时实现收益的最大化。当在 Mesos 集群里运行 Spark 时，资源共享有两种模式：

- 细粒度：Spark 默认会将每个任务当作独立的 Mesos 任务来运行。这意味着 Spark 会和其他框架共享集群资源。在细粒度模式下，Mesos 能够以弹性的方式在所有应用程序间共享集群资源，但是弹性的代价是启动任务时产生更高的额外消耗。这样的额外消耗对于大部分应用程序而言不是很重要，但是会影响时间要求严格的应用程序。

Mesos 0.21 以下版本有 bug，在细粒度模式下，内存使用率过高时，Spark 应用程序无法继续执行（https://issues.apache.org/jira/browse/MESOS-1688）。这个问题的一个解决方式是确保应用程序执行器永远在 slave 上保留至少 32MB 的空闲内存。

- 粗粒度：粗粒度模式会在 Mesos 节点上启动长期运行的 Spark 任务时预留所有资源，然后按需将这些预留资源分配给 Spark 作业。因此，运行在粗粒度模式下时，作业无须额外消耗来完成 Mesos 节点上任务的启动，该模式适用于需要低延时的应用程

序，比如迭代式查询。

通过 spark.mesos.coarse 属性控制模式。通过 SparkConf 将该属性设置为 true，就可以将 Spark 运行在粗粒度模式下，如下：

```
conf.set("spark.mesos.coarse", "true")
```

除了运行模式，在 Mesos 上运行 Spark 时，还有一些其他参数可配置：

- spark.cores.max：该参数指定 Spark 可以获得的最大 CPU 核心数量。默认为能够获取 Mesos 提供的所有资源。可以使用该参数控制资源预留策略。除非 Spark 是唯一运行在 Mesos 上的应用，否则就需要调整上述默认值。通过将 spark.cores.max 参数设置为合理值来限制 Spark 能够使用的资源总量：

  ```
  conf.set("spark.cores.max", "8")
  ```

  上述设置将 Spark 能够使用的内核数设为 8。与之类似，spark.executor.memory 属性控制执行器能够使用的内存总量。

在你的环境中使用合适的模式，通过调整 spark.cores.max 和 spark.executor.memory 的值来控制分配给 Spark 的资源，这会大幅提高集群资源使用率且降低应用程序响应时间。

- spark.mesos.extra.cores：该属性控制粗粒度模式下每个任务能够申请的额外 CPU 核心数量。默认设置为 0。设置了该参数后，能申请到的 CPU 核心数是 offer 内的 CPU 核心数加上额外 CPU 核心数。能申请到的 CPU 核心总量还是由 spark.cores.max 属性控制。
- spark.mesos.executor.home：该属性指定执行器可以访问的 Spark 安装路径。该值默认和 driver 节点上的 SPARK_HOME 相同。该属性只在没有设置 spark.executor.uri 时被使用。
- spark.mesos.executor.memoryOverhead：该属性指定会被加到任务所申请的内存总量（由 spark.executor.memory 指定）上的额外内存消耗。该值默认设为执行器内存 × 0.07，最小值是 384，单位是 MB，其中 0.07 被硬编码到到代码中。最终 overhead 取 spark.mesos.executor.memoryOverhead 和 0.07 × spark.executor.memory 之中更大的值。

Spark 文档（http://spark.apache.org/docs/latest/configuration.html）详细列举了所有 Spark 的配置参数。

小结

本章介绍了 Spark，讨论了如何在 standalone 模式下运行 Spark。同时介绍了在 Mesos 上搭建 Spark 集群的详细步骤。下一章会探讨在 Mesos 上，如何构建复杂的数据分析基础架构。

第4章

Mesos 上的复杂数据分析

本章将介绍多个在 Mesos 上进行复杂数据分析的框架。我们会介绍如何在 Mesos 上搭建 Storm 和 Spark Streaming 来处理实时数据流，以及如何在 Mesos 上运行 NoSQL 数据库 Canssandra。

本章包括如下内容：

- 复杂数据和 Lambda 架构的兴起
- Storm
- Spark Streaming
- Mesos 上的 NoSQL

复杂数据和 Lambda 架构的兴起

大数据的爆炸式增长不仅体现在生产的数据量上，也体现在要求处理海量数据得到有意义结果的速度和多样性上。因此，数据和计算的速度推动开发人员开发实时流处理框架，同时，数据天然的多样性和松散性也促进了NoSQL的演进。

随着**物联网**（**Internet of Things**，**IoT**）的兴起，传感器、社交媒体、机器事务、监控等都在大规模地高速产生数据。这些数据能够提供的信息非常有价值，但是如果数据的分析结果有延迟，或者仅能分析过期数据，那么这些数据就会丧失价值。前一章我们介绍了使用 Hadoop 和 Spark 可以处理的数据量。这些传统工具很适合被用来完成批处理或离线分析，但是它们不是为实时流分析或低延时应用程序设计的，比如，类SQL查询处理。

当流处理在现代数据架构中日益重要时，现代数据架构出现了其他一些组件。现代数据架构包括服务不同需求的不同组件。Lambda 架构（http://en.wikipedia.org/wiki/Lambda_architecture）是十分流行的设计数据架构的方式。它主要包括三层：

1. 批处理层
2. 速度层
3. 服务层

在 Mesos 上运行 Lambda 架构不仅可以共享资源，而且可以帮助容错。批处理层的理念是通过随时处理收集到的数据来导出处理结果的，比如，构建预测模型。前一章已经演示了在 Mesos 上如何使用 Hadoop 和 Spark 进行数据处理。Apache Hama（https://hama.apache.org）是批处理层的另一种框架。它是通用的块同步处理（**Bulk Synchronous Processing，BSP**）框架，在图像处理和矩阵计算领域非常有用。

随着多样化新数据的产生速度不断加快，仅依赖离线模型，周期性地离线处理数据已经不能满足需求。速度层的理念是在数据生成时就完成处理。该层主要进行流处理，也被称为**复杂事件处理**（**Complex Event Processing，CEP**）或实时处理。Mesos 支持 Apache Samza、Apache Storm 和 Spark streaming 框架来实现速度层。Apache Samza（http://samza.apache.org）是构建在 Apache Kafka 框架之上的流处理框架。有项目正在将 Samza 集成到 Mesos 里（https://github.com/Banno/samza-mesos）。下一节会讨论 Apache Storm 和 Spark Streaming。注意不同的流处理框架使用不同的架构，在速度、支持的操作、一致性语义和可扩展性之间采用不同的权衡措施。

服务层负责存储批处理和速度层的输出，并且基于这些输出提供查询服务。Mesos 提供了越来越多的选择，为存储和服务数据构建可扩展层。

- **HDFS（Hadoop 分布式文件系统）**在普通硬件上提供分布式文件系统，第 2 章里有过详细介绍。有项目正试图在 Mesos 上运行 HDFS 来提供高可用的 HDFS。
- Tachyon（http://www.tachyonproject.org）是以内存为中心的存储系统。在 https://github.com/mesosphere/tachyon-mesos 处有 Mesos 上 Tachyon 的原型版本。
- Riak（https://github.com/basho/riak）是分布式的键值存储。有项目正试图让其能在 Mesos 上工作（https://github.com/edpaget/riak-mesos）。
- Elasticsearch（https://elasticsearch.org）是分布式全文本搜索引擎。Elastic Search 可以在 Mesos 上运行（https://github.com/mesosphere/elasticsearch-mesos）。
- Apache Canssandra（http://cassandra.apache.org）是 NoSQL 数据库，下文

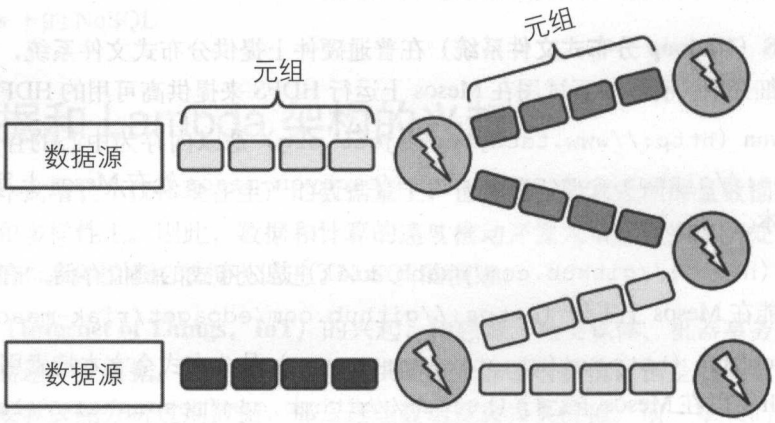
会讲述如何将其运行在 Mesos 上。

除了这三个层次之外，还需要三层之间的连接器来接收数据，发送到其他层。Apache Kafka（http://kafka.apache.org）是分布式发布-订阅（也称为pub/sub）消息系统。Pub/sub系统是现代数据架构的后台机制。它们以松散的方式将不同的数据处理框架连接起来，适用于多种应用场景。有项目正在积极地将Kafka集成到Mesos里（https://github.com/stealthly/kafka-mesos）。另外，随着越来越多的框架使用即将发布的 Mesos 里的持久化存储（https://issues.apache.org/jira/browse/MESOS-1554），Mesos 上的复杂数据处理会越来越健壮。这并不意味着，复杂数据分析需要复杂的工具集，Mesos 支持许多其他架构来满足复杂数据处理的多样化需求。本书撰写时，很多这样的项目还处在早期发展阶段，但是都已经取得了实质性进展。

Storm

Apache Storm 是实时分布式流事件处理引擎（https://storm.apache.org）。Storm 特点是事务性、可靠、可扩展、可容错，并且提供了易用的 API。和 MapReduce 相比，它的架构完全不同。MapReduce 系统，比如 Hadoop、Spark 等都将代码移动到数据附近。这意味着在 MapReduce 架构里，每个节点都有一些数据，每个节点也拥有完全相同的代码来生成结果。但是在 Storm 里，每个节点完成不同类型的处理，处理不同的数据流。

Storm 的主要抽象概念是流过节点的流（元组流），每个节点完成一些处理。元组非常通用，可以包含一系列任意类型的可序列化的对象。在 Storm 里，处理序列用拓扑来描述。拓扑永远在运行，在流数据到达时完成处理。



Storm 拓扑

上图展示了基本拓扑概念。拓扑包含数据源（spout）和数据操作（bolt）。Spout 是数据源。它们侦听数据源，将元组发送到拓扑里。每一次循环代表一次数据操作，完成一些处理，这些流动的元组和数据操作构成的 DAG（有向无环图）就是拓扑。

Storm 是主–从架构。Numbus 是 master 节点守护进程，负责协调和监控。worker 节点运行 supervisor，执行拓扑的一部分。Nimbus 和 supervisor 通过 ZooKeeper 或者本地磁盘相互通信。

Storm有很多高级特性，比如，支持精细监控、事务语义使用 Trident 等，这些不在本书讨论范围。更多Storm 的内容，参考 Quinton Anderson 的《Storm 实时数据处理》。

Mesos 上的 Storm

Nathan Marz 开发了调度器和执行器的第一个版本，随后项目在社区（`https://github.com/mesos/storm`）里进一步发展。在 Mesos 上运行任意框架，都需要调度器来代表框架的任务向 Mesos 申请资源，也需要执行器运行这些任务。如下步骤可以完成 Mesos 上 Storm 的安装：

1. 安装 Mesos。
2. 克隆 storm-mesos 的存储库，并且进入根目录：

```
ubuntu@master:~ $ git clone https://github.com/mesos/storm
ubuntu@master:~ $ cd storm
```

3. 存储库的 `bin` 目录下包含 `build-release.sh` 脚本。该脚本包含很多子命令，可以通过 `-h` 参数查看。先下载未更改的 Apache Storm 发行版。该脚本会下载 `pom.xml` 文件里 `version` 属性指定的版本。默认下载最新版，一般适用于大多数情况。如果需要 Storm 的特定版本，需要将 version 属性设置为所需版本。本书撰写时，默认版本是 `0.9.3`。也可以通过设置 `MIRROR` 环境变量指定下载镜像：

```
ubuntu@master:~/storm $ ./bin/build-release.sh downloadStormRelease
```

4. 上一步命令会下载名为 `apache-storm-VERSION.zip` 的文件。将下载的压缩文件作为脚本的参数：

```
ubuntu@master:~/storm $ ./bin/build-release.sh apache-storm-*.zip
```

5. 上一步命令会在当前目录创建名为 `storm-mesos-VERSION.tgz` 的 `storm-mesos` 发行版。
6. 需要更新 Storm 的配置来匹配集群设置。Storm 使用 YAML 配置文件格式。更新

conf/storm.yaml，添加如下设置：

```
mesos.master.url: "zk://master:2181/mesos"
storm.zookeeper.servers:
        - "master"
nimbus.host: "master"
```

这里，mesos.master.url 参数指定为 <host:pair>，这是 Mesos master 运行的 url。storm.zookeeper.servers 列出 Storm 使用的 ZooKeeper 服务器。nimbus.host 指定 Storm 集群的 master。下节会详细介绍 storm-mesos 的所有配置选项。

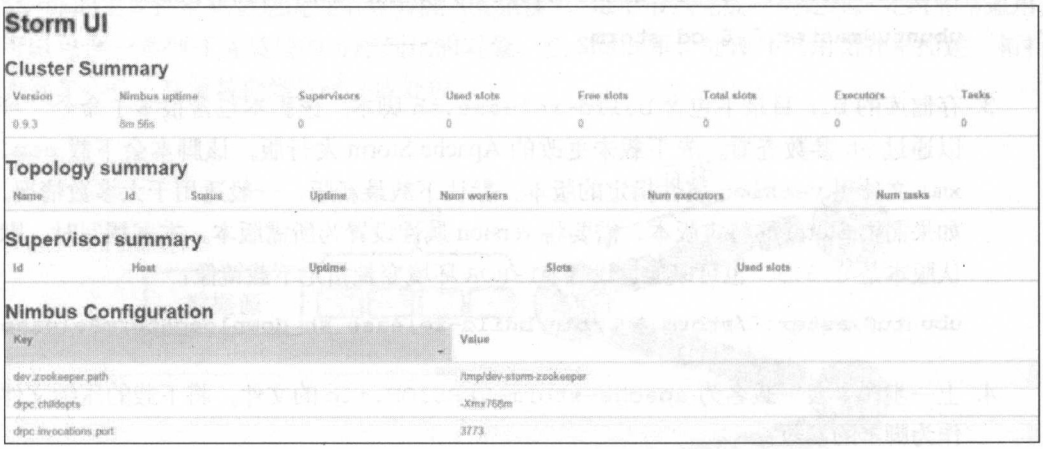
7. 运行如下命令启动 Storm 的 master-nimbus。

```
ubuntu@master:~/storm-mesos $ bin/storm-mesos nimbus
```

8. 也可以选择启动 Storm UI，在 <storm-master:port> 处访问，本例中，是 http://master:8080。

```
ubuntu@master:~/storm-mesos $ bin/storm ui
```

至此，Storm 已经运行在 Mesos 上了。Storm Web UI 如下图所示，显示集群配置有 0 个 supervisor，因为 supervisor 是在运行拓扑时按需创建的。



另外，在 Mesos UI 上，Storm 会被列为活动框架。现在就可以运行多种流处理作业了。Storm 项目在 examples/storm-starter 目录下有丰富的样例拓扑。运行 ExclamationTopology，该拓扑会在输入的单词后加上感叹号。ExclamationTopology 是一个基本拓扑，带有一个数据源 word，两个数据操作 exclaim1 和 exclaim2，以线性方式链接：

```
ubuntu@master:~/storm-mesos $ bin/storm-mesos jar examples/storm
```

```
-starter/storm-starter-topologies-*.jar storm.starter.ExclamationTopology
mytopology
```

注意 Storm 要求给定集群里，拓扑名称是唯一的。

上述命令会将 ExclamationTopology 提交到 Storm 集群，命名为 mytopology。可以使用如下命令行接口验证该拓扑正在运行：

```
ubuntu@master:~/storm-mesos $ bin/storm list
```

还可以使用 Storm Web 接口查看拓扑的各种信息。输出以及其他日志存储在 logs 目录下，感叹号之后会显示 Storm 项目的贡献者。https://storm.apache.org/documentation 可以帮助理解更多 Storm 理念及各种 Storm 命令。

Storm-Mesos 配置

Storm-mesos 使用基于 YAML 的配置。至少需要设置如下配置参数：

- mesos.executor.uri：这是执行器的 URI。
- mesos.master.url：这是 Mesos master 的地址。
- storm.zookeeper.servers：这是 Storm master 使用的 ZooKeeper 服务器。
- nimbus.host：这是运行 Storm nimbus 的主机名。

控制资源配置十分重要，因为这会严重影响到 Storm 的可扩展性和延时。Storm-mesos 使用如下配置调优资源：

- topology.mesos.worker.cpu 和 topology.mesos.worker.mem.mb：这两个参数分别指定每个 worker 节点的 CPU 和内存。默认值分别为 1 和 1000MB。该值必须设置得比 worker.childopts 高 25%，来适应任务的额外消耗。比如，如果 worker.childopts 设置为 -Xmx1000m，那么 topology.mesos.worker.mem.mb 必须至少设为 1250。
- topology.mesos.executor.cpu 和 topology.mesos.executor.mem.mb：这两个参数分别指定每个执行器的 CPU 和内存。默认值分别为 1 和 1000MB。

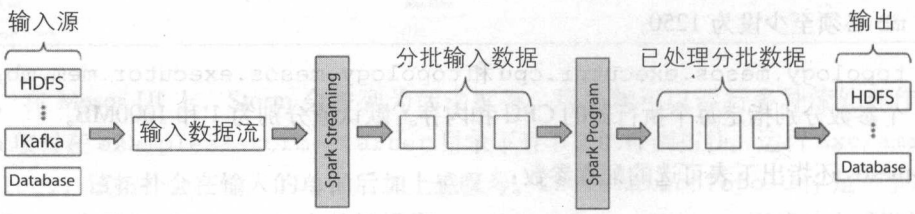
Storm-mesos 还指出下表可选的配置参数：

参数	描述	默认值
mesos.framework.name	指定 Mesos 框架的名称	Storm!!!
mesos.framework.role	指定可以用来认证的框架角色	*
mesos.framework.checkpoint	设定是否启用框架检查点	False
mesos.allowed.hosts（和 mesos.disallowed.hosts）	指定允许（不允许）运行拓扑的主机列表	
mesos.offer.lru.cache.size	指定 offer 的 LRU 缓存大小	1000
mesos.local.file.server.port	用于本地文件服务器的端口	随机端口
mesos.master.failover.timeout.secs	为 master 指定框架故障转移超时时间，以秒为单位	3600
mesos.supervisor.suicide.inactive.timeout.secs	指定 supervisor 在空闲时决定杀掉自己的等待时间，以秒为单位	120

Spark Streaming

我们已经了解到 Spark 可以用来处理大量数据。Spark Streaming 是 Spark API 的扩展，用来处理流数据。它支持各种各样的输入数据源，包括 Twitter、HDFS、Kafka、Flume、Akka Actor、TCP socket 和 ZeroMQ。Spark Streaming 将输入数据流分解成小批量，然后 Spark 程序处理这些离散化的流。已处理的小批量数据可以流转做进一步处理，或者保存到 HDFS、数据库等上。

Spark Streaming 有 DStream 或者离散流（http://www.cs.berkeley.edu/~matei/papers/2012/hotcloud_spark_streaming.pdf）的基础抽象。Spark 内部，DStream 以 RDD 序列的形式存在，DStream 上的操作被转化为 Dstream 里 RDD 的操作。这样自然拥有了 RDD 的所有优势，比如一致性、检查点等。下图展示了 Spark 是如何启动流处理的。



Spark Streaming 架构

Spark Steaming 支持很多不同的操作，无状态和有状态操作都支持。下表列出了目前支持的操作：

Spark Streaming 支持的转化

转化	描述
`cogroup(stream2, [numTasks])`	根据给定键值从数据流里将对应的元组连接到一起。也就是说，当输入是（k，v1）和（k，v2），会返回(k, list¡v1¿, list¡v2¿)
`count()`	以新的 DStream 的形式返回 RDD 源里的元素数目
`countByValue()`	计算 RDD 源里的键值频次，以 DStream 的形式返回（key,frequency）对
`filter(f)`	过滤，返回新的 DStream，包含函数 `f` 返回值为 true 的 RDD
`join(stream2, [numTasks])`	将两个数据流连接到一起，也就是说，输入是（k，v1）和（k，v2）时，会返回（k，(v1，v2)）
`map(f)` 和 `flatMap(f)`	函数 `map()` 对 DStream 里的每个 RDD 执行f函数。`flatMap(f)` 与 `map(f)` 类似，但是可以映射到 0 或者多条记录
`reduce(f)` 和 `reduceByKey(f, [numTasks])`	创建单个元素 RDD 的 DStream，通过 `f` 函数实现值的聚合。`reduceByKey` 还会有额外参数控制并行度
`repartition(numPartitions)`	将 DStream 重新分配，控制并行度
`transform(f)`	将任意的函数 f 应用到 Dstream 里的每个 RDD 上，生成新的 DStream
`union(stream2)`	合并 DStream stream2 里的 RDD 和当前 DStream 里的 RDD
`updateStateByKey(f)`	通过更新函数 f 更新每个键和该键对应的当前状态的值，生成新的 Dstream

Spark Streaming 也支持基于窗口的操作，也就是说可以操作数据的滑动窗。`WindowLength` 和 `slideInterval` 参数控制窗口和操作间隔。下表列出了支持的基于窗口的操作：

Spark Steaming 支持的基于窗口的转化

转化	描述
`countByWindow(windowLength, slideInterval)`	返回 Steam 里的滑动窗内的元素个数
`countByValueAndWindow(windowLength, slideInterval, [numTasks])`	返回新的 DStream，其中键的值和窗口内键的出现频率相同
`window(windowLength, slideInterval)`	根据给定参数创建新的包括窗口内批数据的 DStream
`reduceByWindow(f, windowLength, slideInterval)`	创建新的 DStream，使用函数 f 聚合元素值
`reduceByKeyAndWindow(f, windowLength, slideInterval, [numTasks])` 和 `reduceByKeyAndWindow(f, invFunc, windowLength, slideInterval, [numTasks])`	创建新的 DStream，其中每个键的值是函数 f 处理滑动窗内的元素的输出。也可以传入逆 reduce 函数做增量处理，从而提高效率

和 Spark 类似，这些操作是惰性执行的，如下输出操作才会触发计算：

Spark Streaming 支持的输出操作

输出操作	描述
`foreachRDD(f)`	这是基本输出运算符，必须和其他操作联合使用（比如打印、保存到外部系统等）
`print()`	打印 DStream 里的每个批次数据的前十个元素
`saveAsObjectFiles(prefix, [suffix])` `saveAsTextFiles(prefix, [suffix])` `saveAsHadoopFiles(prefix, [suffix])`	将 DStream 分别保存到序列化文件，文本文件和 HDFS 文件里。文件名是 `prefix-TIME_IN_MS[.suffix]`

在 Mesos 上运行 Spark Streaming

如果已经在 Mesos 上运行了 Spark，那么随时都可以开始使用 Spark Steaming。无须为 Spark Streaming 做任何特殊配置。Spark 发行版在 examples 目录下包含多种样例，包括

Spark Streaming 样例。让我们运行 Spark Stream 样例之一，NetworkWordCount。该样例统计每秒给定网络流里给定单词出现的数量。

首先，需要创建网络流，可以在 TCP 端口使用 Netcat 发送文本。打开终端，键入如下命令，在端口 9999 启动 Netcat 服务器：

```
ubuntu@master:~ $ nc -lk 9999
```

现在在另一个终端，启动 Spark Steaming 样例：

```
ubuntu@master:~ $ cd spark
ubuntu@master:~ $ ./bin/run-example streaming.NetworkWordCount localhost
9999
```

NetworkWordCount 已经在运行了，会侦听第一个终端启动的 Netcat 里的输入。它会打印单词及每秒该单词出现的频率。比如，如果我们在 Netcat 窗口键入 hello world，就会在 Spark Streaming 窗口看到如下输出：

```
(hello,1)
(world,1)
```

Spark Streaming 调优

Spark Streaming 在 Mesos 里开箱即用。但是，对系统进行调优以满足实时流处理的需求和优化资源的使用是至关重要的。如下是调优需要考虑的几个方面。

选择批量大小

选择流数据的批量大小是能否及时处理输入数据的决定性因素。批量大小不能设置过小，否则集群资源可能被浪费。另一方面，如果批量大小设置过大，流计算可能跟不上。因此，推荐的方式是从对于应用程序而言比较保守的批量大小开始，然后逐步检测出更小的数值。如果每个批次数据端到端处理的速度能够比输入批次数据的速度快，那么就说明系统可以胜任当前的处理速度。要度量该时间，可以使用 Spark 提供的org.apache.spark.scheduler.StreamingListener 接口。持续增加的延时是系统不能处理当前数据的信号。

垃圾回收

在生产环境运行 Spark Streaming 时，需要格外注意的重要配置参数是 `spark.cleaner.ttl`。该参数控制 Spark 记忆的元数据的长度。Spark 默认不会移除任何元数据，有太多元数据时，就会影响到流处理应用程序。另外，如果该值设置得过低，窗口操作可能就无法处理窗口长度内的 RDD。因此，`spark.cleaner.ttl` 的值必须设置得比流处理应用程序的最大窗口长度更大。如果没有设置 `spark.cleaner.ttl`，Spark 会用最近最少使用（LRU）的方式清除所有 RDD。另外，将 `spark.streaming.unpersist` 设置为 `true` 可以启动一种更为智能的反持久化方案，系统会计算出哪些 RDD 可以从内存里移除。另外，推荐使用 Java 虚拟机的并发标记和移除式垃圾回收器，因为这样允许很多小型的 GC 暂停，而不是一个大型的，这会让流处理延时更稳定。

并发

使用可用集群资源并行化处理十分重要。必须给操作传输合适的 `numTask` 参数，默认值是 8。也可以通过 `spark.default.parallelism` 来改变该默认值。

故障处理

必须考虑到如果 driver 节点和 worker 节点发生故障时该如何处理，因为所有的中间数据都可以根据 RDD 处理链重新计算出来。为了确保 driver 节点的恢复，必须启动检查点（通过 `ssc.checkpoint` 参数），应用程序必须检查前一个检查点状态是否存在。如果输入源是网络链接而 worker 节点在复制前失败了，还可以将数据复制到其他节点上，但是可能会丢失一小部分数据。对于持久化输入存储，比如 HDFS，worker 节点失败不会造成任何数据的丢失。Spark Streaming文档（`http://spark.apache.org/docs/latest/streaming-programming-guide.html`）详细讨论了 Spark Streaming 所提供的容错语义。

任务额外开销

Mesos 任务启动的额外开销对于低延时应用程序，比如 Spark Streaming，可能是致命的。Spark Streaming 必须运行在细粒度 Mesos 模式下，来减少任务启动的额外开销，第 3 章对此有详细解释。另外，为了减少 GC 暂停，Spark Streaming 将 RDD 以序列化二进制的格式持久化存储下来。这样，序列化/反序列化的额外开销可能很大，推荐使用快速序列化框架，比如 Kryo（`https://github.com/EsotericSoftware/kryo`）。另外，序列化任务也可能减少任务的网络传输时间，从而降低任务启动的额外开销。

Mesos 上的 NoSQL

数十年里，SQL 一直是数据分析的主要工具。随着大数据的兴起，很多系统尝试将数据库应用到大规模复杂数据分析领域。这样的系统包括 Hive、Shark、Spark SQL 和 NoSQL 数据库，比如 Cassandra、Hypertable 等。可以将这些类型的工作负载都运行在 Mesos 上，同时利用 Mesos 的优势，包括资源共享、容错等。下节详细介绍在 Mesos 上安装 Cassandra 的步骤。

Mesos 上的 Cassandra

Apache Cassandra（`http://cassandra.apache.org`）是流行的 NoSQL 数据库。Cassandra 由 Facebook 发起，在很多大规模部署环境上起着重要作用。通过在 Mesos 上运行 Cassandra，可以利用 Mesos 的容错和扩展能力。Cassandra 非常适合 Mesos，因为其架构是完全去中心化的。

要想在 Mesos 上运行 Cassandra，需要调度器和 Mesos 协调 Cassandra 所需的资源，执行器实际运行 Cassandra 的守护进程。调度器还需要将所有发行版和配置文件复制到所有节点上。Cassandra 配置需要定制种子节点，一旦调度器接受了来自 Mesos 的 offer，种子节点就会包含到调度器里。下列是在 Mesos 上运行 Cassandra 的步骤：

1. 安装 Mesos。
2. 登入 Mesos master，从 Mesosphere 下载最新的预构建的 Cassandra-mesos 发行版。本书撰写时，最新版是 2.0.5：

```
ubuntu@master:~ $ wget http://downloads.mesosphere.io/cassandra/cas
sandra-mesos-2.0.5-1.tgz
```

3. 解压缩该文件，`cd` 到目录下：

```
ubuntu@master:~ $ tar xzf cassandra-mesos-*.tgz
ubuntu@master:~ $ cd cassandra-mesos-*
```

4. 需要编辑 `conf/mesos.yaml` 来反应集群配置。默认配置针对和 ZooKeeper 一起运行的本地 Mesos 集群。下表列出配置选项：

Cassandra-mesos 配置选项

选项	描述	默认值
mesos.executor.uri	指定 Cassandra 执行器的位置	http://downloads.mesosphere.io/cassandra/cassandra-mesos-2.0.5.tgz
mesos.master.url	指定 Mesos master 位置	zk://localhost:2181/mesos
java.library.path	指定 Mesos 库的位置	/usr/local/lib/libmesos.so
cassandra.noOfHwNodes	指定想要 Cassandra 运行在多少台机器上	1
resource.cpus resource.mem resource.disk	每个节点上 Cassandra 的资源要求	0.1 2048 1000

5. 在 Mesos 调度器上启动 Cassandra：

```
ubuntu@master:~/cassandra-mesos $ bin/cassandra-mesos
set -o errexit -o pipefail

FRAMEWORK_HOME='dirname $0'/..
dirname $0

export MESOS_NATIVE_LIBRARY=$(sed -e 's/:[^:\/\/]/="/g;s/$/"/g;s/*=
/=/g' "$FRAMEWORK_HOME"/conf/mesos.yaml | tr -d $'\'' | grep -v \#
| grep java.library.path | sed 's/java.library.path=// g;s/"//g')
...
# Start Cassandra on Mesos
...
0 [main] INFO mesosphere.cassandra.Main$ - Starting Cassandra on
Mesos.
...
114 [Thread-0] INFO mesosphere.cassandra.CassandraScheduler -
Starting Cassandra cluster ${clusterName} for the first time.

Allocating new ID for it.
```

```
...
I0429 19:36:36.742849 27508 sched.cpp:391] Framework registered
with 20140429-193514-580538634-5050-25835-0000
175 [Thread-1] INFO mesosphere.cassandra.CassandraScheduler -
Framework registered as 20140429-193514-580538634-5050-25835-0000
437 [Thread-2] INFO mesosphere.cassandra.CassandraScheduler - Got
new resource offers ArrayBuffer(slave1)
455 [Thread-2] INFO mesosphere.cassandra.CassandraScheduler -
resources offered: List((cpus,2.0), (mem,6489.0), (disk,7935.0),
(ports,0.0))
455 [Thread-2] INFO mesosphere.cassandra.CassandraScheduler -
resources required: List((cpus,0.1), (mem,2048.0), (disk,1000.0))
464 [Thread-2] INFO mesosphere.cassandra.CassandraScheduler -
Accepted offer: slave1
...
```

这里，被截断的输出显示 Cassandra 集群已经注册到 Mesos 上了，并且从 `slave1` 接收到了资源 offer。现在 Cassandra 已经启动并运行了，在 Web UI 里应该能看到列出的框架里有 `Cassandra Test Cluster`。可以通过 Cassandra 查询语言（**Cassandra Query Language (CQL)**）shell 与之交互。从命令行或者通过 Web UI（UI上的`Host` 字段），选择运行 Cassandra 的任意主机，使用如下命令连接到 CQL 会话中：

```
ubuntu@master:~/cassandra-mesos $ bin/cqlsh <cassandra-host>
```

应该能够看到 Cassandra 提示符（`>cqlsh`）。现在就可以运行 Cassandra 查询了。另外注意 Mesos 上的 Cassandra 支持很多场景和功能（比如扩展性），本文对此没有深入介绍。

小结

批量处理系统不再是开发人员可用的唯一数据处理工具，新的应用程序及要求不同类型数据的分析用例层出不穷，而不是只有传统ETL工具和框架，比如 Hadoop，支持的一种场景。因此，与其尝试让批量处理系统更快或扩展传统数据库来处理非结构化数据，倒不如使用正确的工具来完成这些工作，从而让这些应用程序的开发和扩展更为容易。

本章探讨了使用运行在 Mesos 上的 Storm 和 Spark Streaming 处理实时和流数据的可选方案。也讲解了如何使用运行在 Mesos 上的 Cassandra 实现更多探索性数据分析。Cassandra 也是 Mesos 上更为通用的应用程序实例之一。本书后面的章节会探索 Mesos 的内部工作机制及其运维领域问题，下章会先讨论各种调度框架。

第5章 在 Mesos 上运行服务

前一章讨论了在 Mesos 上如何运行多种数据处理框架，企业环境里的另一个重头戏就是服务。每个企业都运行着各种各样的服务，这些服务对企业的成功起着重要的作用。本章会介绍服务及在 Mesos 上运行服务所用的不同的调度器框架：

- 服务的介绍
- Marathon
- Chronos
- Aurora
- 服务发现
- 打包 chronos

服务的介绍

传统的企业运行着很多长期运行的服务来支撑日常运营，比如 Web 服务、数据流水线等。服务是一个自包含、独立部署并管理的功能单元。**面向服务架构（SOA）**及最近兴起的微服务架构，鼓励将松耦合的服务组合成应用程序。越来越多的现代应用程序由多个微服务组成，这么做有很多好处，比如，代码可重用、易于扩展、故障隔离、多平台支持、灵活部署及高度敏捷。

之前我们已经了解到 Mesos 很擅长处理批量、实时等处理框架，这些框架的作业一般都是短暂的。企业基础架构上运行着很多天生就需要长期运行的应用和服务，这里的需求和数

据处理框架很不一样。这些长期运行的服务不仅是业务的核心，而且消费绝大部分基础架构的资源。因此，在 Mesos 上运行服务的能力至关重要。

要想大规模运行服务，基础架构必须能够支持如下方面：

1. 部署关心的问题是给定服务应该运行在哪里。如果服务依赖于其他服务，那么其部署可能会很复杂，因为所依赖的服务可能限定了服务能够部署的位置。
2. 配置管理和打包是为了确保满足服务所需的依赖条件，在服务启动之前就在环境里为该服务完成了正确的配置。
3. 当多个服务实例同时运行时，服务发现和负载均衡就变得很重要。服务发现回答特定服务实例运行在哪里的问题，而负载均衡则有关于决定给定请求应该送达到哪个实例。
4. 一旦服务被部署，就需要对服务进行健康检查。健康监控信息可用于决定之后的操作，比如服务是扩容还是缩容，或者在故障发生时重新加载。
5. 可用性需求要求服务在高负载和发生故障时仍然可用。

如果将 Mesos 比作数据中心的内核，在大规模基础架构上，服务调度器就是其次最重要的组件。虽然 Mesos 上较早完成开发的架构绝大多数都是数据处理相关的，但是对于企业环境而言，服务的运行也必不可少。有相当比例的应用程序需要编写或者重写之后，才能够使用 Mesos 提供的分布式基础架构，但是也有很多应用只需在节点上开始执行，在故障时重启就可以了。调度器框架可以使其达到这个目标。服务调度器也被称为元框架或者元调度器，因为它们在基础架构之上运行的应用程序间起着中介器的作用。服务调度器负责运行服务、处理故障的复杂性、重启服务，以及按需扩容服务。

Marathon

Marathon 是在 Mesos 上运行长期服务的框架（`https://github.com/mesosphere/marathon`）。这些服务都要求高可用，也就是说 Marathon 必须能够在某台机器发生故障时自动在其他机器上启动服务实例，也必须能够弹性扩展。这些服务包括可以运行的标准 shell 及特别的执行器。Marathon 好比数据中心或集群的 `init.d`，可以确保在其上运行的服务一直运行着。Marathon 就是为运行任务且保证这些任务能持续运行而设计的。Marathon 可以运行其他框架，比如 Hadoop 及其自身。

Marathon API

Marathon 拥有非常完备的 REST API 来管理服务的生存周期及 Java、Scala、Python 和 Ruby 的各种客户端：

类型	端点	描述
应用	`GET /v2/apps`	列出所有运行着的应用程序。它还可以接受 `cmd` 参数，这样就可以列出所有匹配 `cmd` 运行结果的应用程序
	`GET /v2/apps/appid`	列出应用 ID 为 `appid` 的应用程序
	`GET /v2/apps/appid/versions`	列出该应用程序的所有版本
	`GET /v2/apps/appid/versions/version`	列出该应用程序特定 `version` 的配置
	`GET /v2/apps/appid/tasks`	列出该应用程序的所有任务
	`GET /v2/tasks`	列出所有应用程序的所有任务
	`POST /v2/apps`	启动一个新的应用程序。应用程序的所有信息以 JSON 格式定义在请求里
	`POST /v2/apps/appid/restart`	重启特定应用程序的所有任务
	`PUT /v2/apps/appid`	修改特定应用程序未来任务的配置
	`DELETE /v2/apps/appid`	销毁应用程序
	`DELETE /v2/apps/appid/tasks`	强行终止应用程序的所有任务。如果指定了主机参数，那么只强行终止这些主机上的任务。如果 scale 参数设为 true，应用程序的配置会被更新，实例数会减掉被该请求强行终止的任务数
	`DELETE /v2/apps/appid/tasks/taskid`	杀死应用程序里特定 `taskid` 的任务。如果 scale 参数设为 true，请求成功之后，应用程序实例数会减 1

（续表）

类型	端点	描述
组	GET /v2/groups	列出所有组
	GET /v2/groups/groupid	列出指定 groupid 的组
	POST /v2/groups	创建并启动新的应用程序组
	PUT /v2/groups/groupid	改变已部署应用程序组的参数
	DELETE /v2/groups/groupid	销毁指定组
部署	GET /v2/deployments	列出所有正在进行的部署
	DELETE /v2/deployments/deploymentid	取消指定 deploymentid 的部署
服务器信息	GET /v2/info	显示当前 Marathon 实例的 info
事件订阅	GET /v2/eventSubscriptions	列出所有事件订阅的回调 URL
	POST /v2/eventSubscriptions	为事件订阅注册回调 URL，URL 作为参数传入
	DELETE /v2/eventSubscriptions	为事件订阅取消已注册的回调 URL，URL 作为参数传入
队列	GET /v2/queue	列出任务队列的内容
其他	GET /ping	返回 ping 的状态
	GET /logging	显示 logging 状态
	GET /metrics	从 Marathon 返回 metrics
	GET /help	打印 help

运行 Marathon

按照如下步骤运行 Marathon：

1. 安装 Mesos。

2. 下载 Marathon 发行版。可以按照文档直接从源码构建。本书撰写时，最新版是 0.8：

```
ubuntu@master:~ $ wget http://downloads.mesosphere.io/marathon/v0.8
.0/marathon-0.8.0.tgz
```

3. 解压缩发行版，cd 到其根目录：

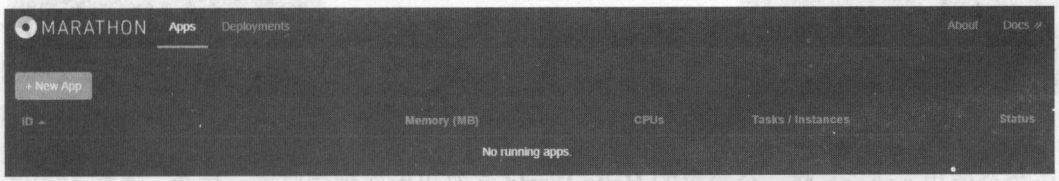
```
ubuntu@master:~ $ tar xzf marathon-*.tgz
ubuntu@master:~ $ rm marathon-*.tgz; cd marathon-*
```

4. 使用脚本启动 Marathon。需要传入 Mesos master 和 ZooKeeper 地址：

```
ubuntu@master:~/marathon $./bin/start --master zk://master:2181/mesos
--zk master:2181
```

Marathon 提供了很多选项来满足不同的需求，命令行参数的完整列表见 http://mesosphere.github.io/marathon/docs/command-line-flags.html。

5. 脚本还会在 8080 端口启动 Marathon Web UI：

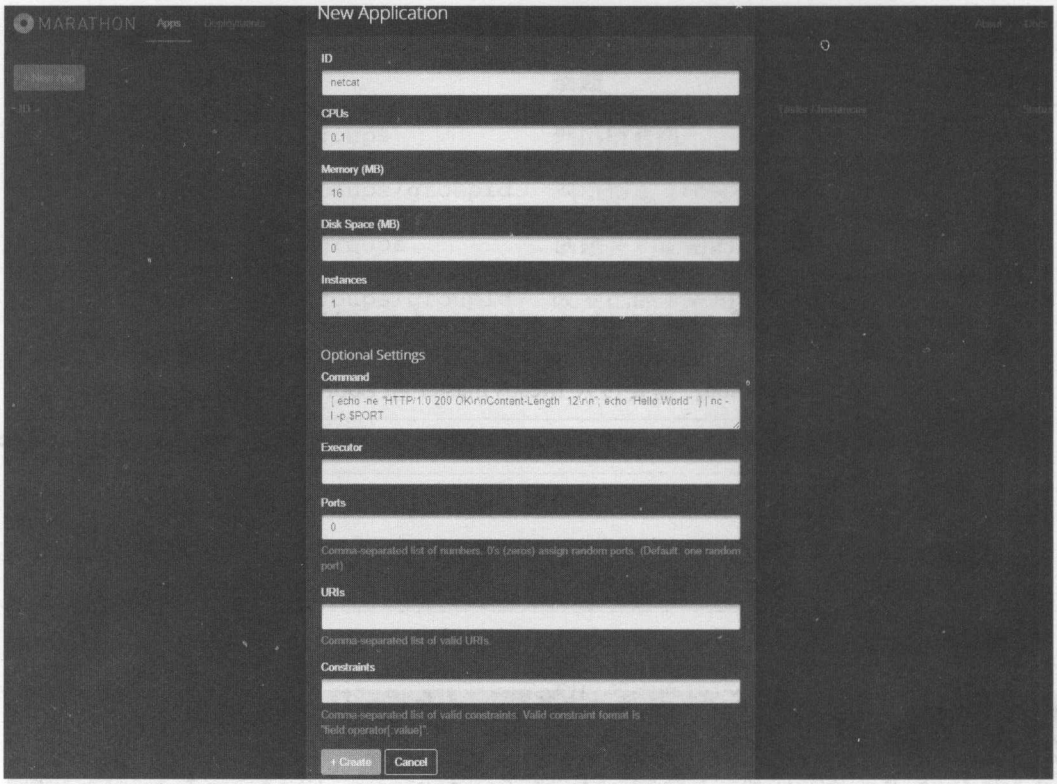


Marathon 样例

Marathon Web UI 提供了使用 Marathon 的便捷接口。可以运行一个非常简单的 Web 服务器来验证安装是否正确。创建新的 App，并且提供任务所需的资源及如何运行任务的信息。在 ID 字段提供应用程序的名称。对于这个简单任务来说，0.1CPU 和 10MB 内存就足够了。

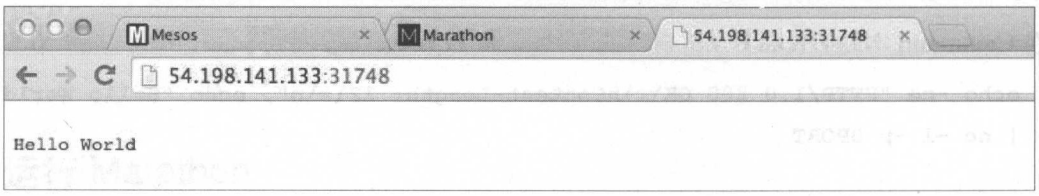
在 **Command** 字段插入如下字符串：

```
{ echo -ne "HTTP/1.0 200 OK\r\nContent-Length: 12\r\n"; echo "Hello World";
} | nc -l -p $PORT
```



该命令使用 `netcat` 将一个简单的 **Hello World** 字符串发送到浏览器。注意，选择 `$PORT` 变量指定的端口。Marathon UI 会显示作业的状态。可以从 Web UI 上选择扩展该作业。另外，Mesos Web UI 也会显示与之对应的 Mesos 作业的任务正在运行。注意查看该任务运行在哪个主机和端口上，这样就可以在服务器上加以验证。还可以通过 Marathon REST API 得到这些信息。该命令在处理单次请求后就会结束。

因此，Marathon 会在每次接收到请求后创建一个新任务，来保证我们所实现的上述请求服务持续可用。



Marathon 发行版在 `examples` 目录下包含很多样例作业。注意，使用 Marathon REST API 代替 Web 接口时可能会启动同名作业。如下，`hello_world.json` 定义了样例服务器：

```
{
  "id": "netcat",
  "cmd": "{ echo -ne "HTTP/1.0 200 OK\r\nContent-Length: 12\r\n"; echo
"Hello World"; } | nc -l -p $PORT",
  "mem": 10,
  "cpus": 0.1,
  "instances": 1,
}
```

可以使用 `hello_world.json` 和 Marathon REST API 启动样例服务器：

```
ubuntu@master:~ $ curl -X POST -H "Content-Type: application/json"
http://master:8080/v2/apps -d@hello_world.json
```

下文继续探讨 Marathon 的其他特性。

约束条件

调度器最重要的功能之一是控制在哪里及如何产生任务。Marathon 里的约束条件允许限制特定应用程序运行的位置，因此可以受益于位置、容错或其他约束。启动应用程序的同时约束条件会被强制执行。一个约束包括字段、运算符及可选参数。字段是分布在 Mesos slave 或者其主机名上的任意属性。Marathon 可以通过命令行的 `--constaint` 参数或 REST 接口上的约束字段来指定约束条件。

有三种运算符可以用来指定约束条件：

- UNIQUE：指定某字段的唯一性。比如，在每个主机上只运行一个实例/任务。
- CLUSTER：将任务限定到拥有特殊属性的节点上。该运算符在强制本地化，特别是h/w时很有用。
- GROUP_BY：将任务在给定字段（rack/数据中心）上平均分配。可选参数可以指定尝试运行组的最小数目。
- LIKE：该运算符允许基于字段里的正则表达式来过滤主机。
- UNLIKE：该运算符和 LIKE 运算符相反。它过滤出和给定正则表达式不匹配的主机。

事件总线

Marathon 的内部总线会抓取 Marathon 的所有事件，比如，API 请求、扩展事件，等等。可以用来集成负载均衡器、监控和与 Marathon 一起使用的其他外部系统。可以通过指定 `--event_subscriber <subscription>` 在服务器启动时开启该功能。订阅是可插式接口，默认为 HTTP 回调订阅者。会向 `--http_endpoint` 指定的主机 POST JSON 格式的事件。事件总线里会有如下事件：

- API 请求，应用程序的创建/更新/删除请求
- 每次任务状态改变时接收到的状态更新
- 为每个框架消息服务的框架消息事件
- 添加或删除新的事件订阅者时触发的事件订阅事件
- 健康检查事件由多种健康检查事件触发，比如，添加、删除、故障或者状态改变
- 部署事件，为各种部署事件生成，比如多种部署的成功或故障

artifact store

artifact store 是部署应用程序使其运行所需的特定资源的存储地，比如特定的文件。运行在 Marathon 上的应用程序可以在其应用定义里使用 `storeUrls` 字段。`storeUrls` 是 URL 的列表。每个 URL 被下载并存放进 artifact store。一旦下载完成，该 artifact 的路径会被添加到应用程序定义里的 `uris` 字段下。内容的路径是唯一的，并且不会再次下载，这样能够保证应用更快地运行。artifact store 有自己的 REST API，允许在 artifact 上做 **CRUD（创建、更新、读取、删除）**操作。artifact API 支持通过自动化来简化部署过程。artifact store 能够运行在多种后台存储系统上，包括本地文件系统和 HDFS。可以使用参数 `--artifact_store` 来配置 Marathon 使其使用 artifact store，该参数提供 artifact store 的位置路径。注意 artifact store 的所有功能都可以在没有 artifact store 的情况下手动完成，因此，artifact store 的使用是可选的，不是必需的。

应用组

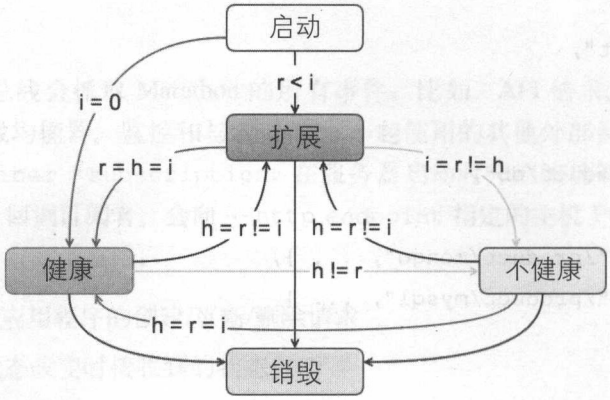
应用组将应用分为互斥的交集以便管理。组使得管理逻辑上相关的应用程序变得简单，并且允许在整个组的范围内执行像扩展这样的操作。比如，将组扩展两倍会将该组的所有应用都加倍。组可以包括应用程序或者组。组是分层的，可以用相对或绝对路径来表示。因此组是可以由组组成的。如下是组定义的示例：

```
{
  "id": "/product",
  "groups": [
    {
      "id": "/product/db",
      "apps": [
        { "id": "/product/mongo", ... },
        { "id": "/product/mysql", ... }
      ]
    },{
      "id": "/product/service",
      "dependencies": ["/product/db"],
      "apps": [
        { "id": "/product/rails-app", ... },
        { "id": "/product/play-app", ... }
      ]
    }
  ]
}
```

Marathon 在启动、停止、升级及扩展时会考虑依赖关系。依赖关系可以定义在应用程序或者组级别上。定义在组级别的依赖关系适用于该组的所有成员（组或应用程序）。

应用程序健康检查

对于长期运行的服务而言，详细的健康检查功能必不可少。Marathon 的默认健康检查任务认为只要 Mesos 任务的状态为 `TASK_RUNNING`，其应用状态就是健康的。这样简单逻辑的健康检查对于简单应用来说可能已经足够了，但是很多应用程序都有不同的语义来定义何时认为其状态是健康的，以及何时需要重启。Marathon 里的应用程序生命周期如下图所示，图中，`i` 是请求的实例个数，`r` 是运行着的实例个数，`h` 是健康的实例个数，箭头上的标签说明在哪种条件下会触发状态间的转变：



在任务启动时，健康检查就会立刻随之启动，但是会忽略初始故障 `gracePeriodSeconds`。如果在 `maxConsecutiveFaliures` 之后任务还不在健康状态，该任务就会被强行终止。目前，Marathon 支持 HTTP 和 TCP 健康检查，其他选择仍在开发中。

Chronos

Marathon 使得服务能够长期运行。基础架构里另外重要的一环就是循环作业。很多企业运行着几十个作业，它们无须一直运行，但是需要反复地在特定时间触发，比如，备份、**ETL（提取转换加载）**作业、运行其他基础架构，等等。

通常使用 cron 工具和 shell 脚本来实现这样的重复执行。不过这样做不仅容易出错、很难维护，而且并不可靠。如果 cron 作业运行的节点发生故障，相应的作业就不会被执行，从而可能破坏整个企业的工作流。Chronos（`https://github.com/mesos/chronos`）是处理依赖关系的可容错的作业调度器，基于 ISO8601，就像为 Mesos 数据中心内核定制的 cron 一样。ISO8601 是一种国际标准，用来表示并交互日期及时间相关的数据。Chronos 使用 ZooKeeper 来实现容错，依赖 Mesos 来执行作业。Chronos 允许用户运行 shell 脚本，同时也支持依赖关系和重试。

Chronos REST API

Chronos 提供了 REST API 来实现完整的作业管理和监控。Chronos REST API 可以帮助自动化，也被 Web UI 使用。下表列出了一些重要的 Chronos API：

端点	描述
`GET /scheduler/jobs`	列出所有作业。结果是 JSON 文件，包含执行器、invocationCount 及 schedule/parents
`DELETE /scheduler/jobs`	删除所有作业
`DELETE /scheduler/task/kill/jobName`	为指定的作业删除任务
`DELETE /scheduler/job/jobName`	基于 jobName 删除特定作业
`PUT /scheduler/job/jobName`	手动启动一个作业
`POST /scheduler/iso8601`	添加新的作业。传入的 JSON 必须包含作业的所有信息，包括该作业的所有依赖关系
`GET /scheduler/graph/dot`	以 dot 文件的格式返回依赖关系图

Chronos 使用 JSON 来描述作业，包含如下字段：

- name：指定作业名称
- owner：指定该作业拥有者的邮件地址
- command：指定需要 Chronos 运行的命令
- schedule：为作业指定 ISO8601 格式的计划，包括被“/”符号分隔的三部分：
  - 该作业需要运行的次数。指定为“R”，就会永远重复执行该作业。
  - 该作业的启动时间。若没有指定启动时间，该作业会立刻启动。
  - 两次运行间隔。
- async：检查作业是否在后台运行。
- epsilon：指定可以启动作业的时间间隔，以防 Chronos 错过调度时间。格式也是遵守 ISO8601 的符号定义。

如下示例 JSON 指定名为 `HelloWorldJob` 的作业。每次运行时都会在 `/tmp/HelloWorkOut` 文件后加上 **Hello World** 字符串。从 2015 年 2 月 1 日开始，该作业每 2 秒运行 10 次，一次运行可能会延时 15 分钟：

```
{
  "schedule": "R10/2015-02-01T05:52:00Z/PT2S",
  "name": "HelloWorldJob",
  "epsilon": "PT15M",
  "command": "echo 'Hello World' >> /tmp/HelloWorldOut",
```

```
  "owner": "me@example.com",
  "async": false
}
```

运行 Chronos

Chronos 在 bin 目录下有很多帮助脚本，比如，wrapper 脚本可以传输文件并在远程机器上执行这些文件。Chronos 发行版里还包含安装 Mesos 的脚本：

1. 安装 Mesos。
2. 按照 Github 上的文档，可以从源码构建 Chronos。也可以使用预构建版本，位于：

   ```
   ubuntu@master:~ $ wget http://downloads.mesospher.io/chronos/chronos
   -2.1.0_mesos-0.14.0-rc4.tgz
   ```

3. 解压缩并 cd 到根目录：

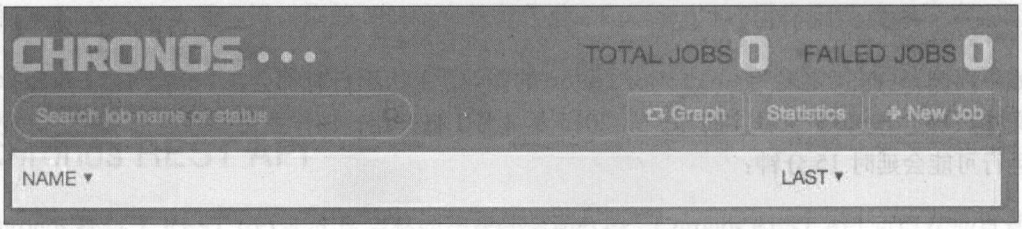
   ```
   ubuntu@master:~ $ tar xzf chromos-*.tgz
   ubuntu@master:~ $ cd chronos
   ```

4. 使用 bash 脚本启动 Chronos。要求传入 Mesos master 和 ZooKeeper 主机地址：

   ```
   ubuntu@master:~ $./bin/start-chronos.bash --master zk://localhost:
   2181/mesos --zk_hosts zk://<Mesos>:2181/mesos --http_port 8081
   ```

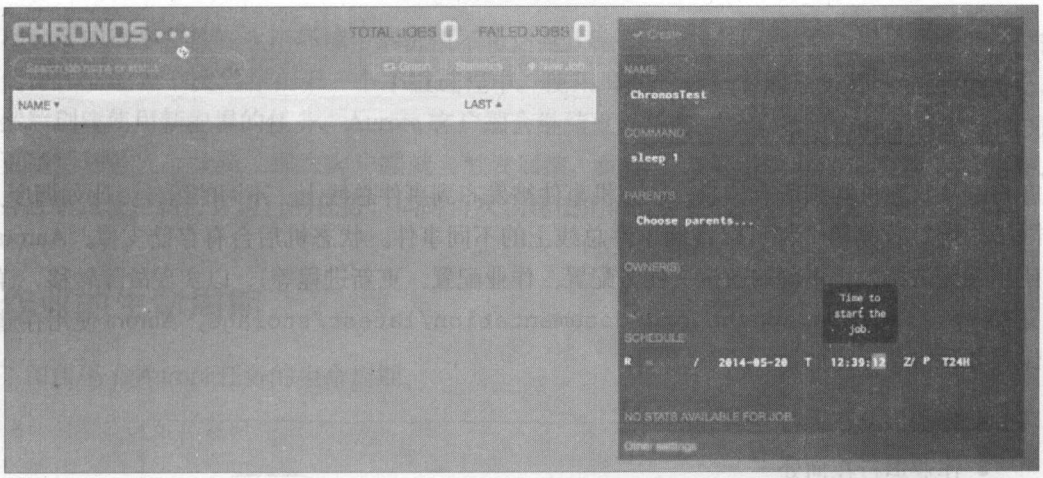
   注意 `start-chronos.bash --help` 会列出 Chronos 支持的所有配置选项。默认配置下，Chronos 运行在 8080 端口。本例运行在 8081 端口。

5. Chronos 启动完成，并且注册为 Mesos 上的框架之一。同时也在 8081 端口启动了 Web UI。



Chronos 样例

本节创建一个在预订时间运行的简单作业。在 Web UI 里，点击 **+ New Job** 按钮，会弹出作业信息的仪表板。计划每天运行一个简单的 sleep 命令，选择触发该作业的时间及作业执行的周期。注意，时间是 UTC 格式的。如果想要该作业在其他作业完成后才触发，而不是按照时间触发，可以在父字段里设定其他作业。一旦作业创建完成，会被列为 `fresh`。一旦作业被排期，状态就会转变为 `success` 或 `failure`。该任务也会出现在 Mesos UI 上，可以从 Mesos UI 访问 `stdout`、`stderr` 和 `logs`。



也可以使用 Chronos REST API，提供合适的 JSON 来创建相同的作业。

```
ubuntu@master:~ $ curl -L -H 'Content-Type: application/json' -X POST -d
'{<job-config>}' chronos-node:8080/scheduler/iso8601
```

Chronos 提供了很多有用的工具和功能，可以调度非常复杂的作业流水线，本书不作深入探讨。

Aurora

Apache Aurora（`http://aurora.apache.org`）是 Apache Mesos 上用来调度作业的另一个服务框架。它提供了很多基本功能帮助快速部署及扩展无状态服务，同时提供良好的容错保障。Aurora 是功能丰富的作业调度框架，能很好地支持海量作业和大量并发用户。它是由 Twitter 开发的，在成为 Apache 项目之前就已经是很多公司的核心基础框架。

Aurora 作业包括多个相似的任务副本。一个 Aurora 任务就是在同一台机器上运行的一个或者一系列进程。因为 Mesos 只提供任务的抽象，Aurora 提供了一个名为 Thermos 的组件在单个 Mesos 任务里管理多个进程。因为每个任务都是运行在沙箱里，Thermos 进程可以在这些沙箱间共享状态。Thermos 使得可以执行任意进程，并且提供了可观察/监控的接口。Thermos 主要分为两部分。执行器负责启动和管理任务，观察器负责监控后台程序，提供运行任务的信息。Aurora 架构包括如下组件：

- 客户端
- 状态机
- 事件总线
- 存储
- 事件总线订阅者

客户端从状态机得到所有信息。状态机事件被发布到事件总线上。不同的组件，比如调度、限流、统计收集等，都可以订阅事件总线上的不同事件。状态机后台有存储支撑。Aurora 使用存储来持久化不同的数据（任务配置、作业配置、更新进程等），以实现故障转移，详见 `http://aurora.apache.org/documentation/latest/storage`。Aurora 使用存储保存各种信息，比如：

- 哪些作业在运行
- 作业运行在何处
- 作业的配置
- Mesos master、资源配额、锁定及其他元数据

从高层看，Aurora 里的存储架构包括四个主要组件：

- 易失性存储，负责将数据缓存到内存里。Aurora 使用 H2 数据库实现易失性存储。
- 日志管理器，是 Mesos 复制日志的接口。日志管理器使用 Thrift 以二进制数据格式存储。
- 快照管理器，负责检查数据到 Mesos 复制日志里。这有助于加速恢复过程。
- 备份管理器，将快照备份到备份文件里，以备 Mesos 日志丢失或损坏时使用。

所有存储的数据都是不可变的，任何改变其实都是存储了该数据的新版本。存储接口提供了三种操作：

- consistentRead 提供了使用读锁得到数据的一致版本的方法。
- weaklyconsistentRead 提供了不用锁获得数据的方式，不过结果有可能是不一致的。

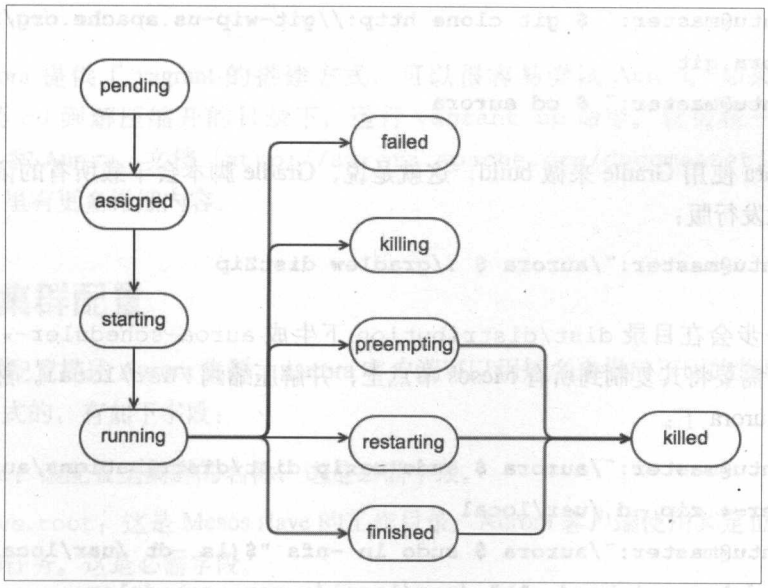
- 所有的写入都是用写锁序列化写入的，首先存储到复制日志里，然后再存储到易失性存储里。这种预写日志技术用来确保一致性，也就是说，写入操作成功要求写入复制日志和易失性存储都成功。

Aurora 用主-从架构来确保高可用，slave 根据日志写入，在故障发生时可以转变为 master。Aurora 通过 ZooKeeper 找到 master。quorum 大小必须是运行着的实例数目的大多数，可以使用参数 `-native_log_quorum_size` 指定。故障发生时，Aurora 会等着外部服务来重启 Aurora 进程。外部工具，比如 Monit，可以用来重启 Aurora 进程。Aurora 将服务注册到 ZooKeeper 上，这样可以用来完成服务发现。

Aurora 的很多特性在生产环境中的价值不可估量。滚动式更新就是其中之一。滚动式更新使得作业配置一次只存在于一个小组进程中。使用旧配置运行着的任务都会被强行终止，然后创建使用新配置的任务。Aurora 客户端会继续更新任务到所需数目。如果一定比例的更新后任务发生故障，那么客户端就会触发回滚。回滚或者更新的取消都会恢复旧配置，并且销毁使用新配置运行的任务，同时再次创建使用旧配置的任务。

作业的生命周期

下图展现了 Aurora 任务的生命周期。



Aurora 任务生命周期

新任务从 **pending** 任务启动。调度器尝试找到满足任务约束条件的机器。一旦找到了满足条件的机器，任务状态就会变为 **assigned**。slave 从调度器接收到任务的配置，生成执行器。一旦调度器从 slave 处收到确认信息，任务的状态就会随之变为 **starting**。任务完成了初始化后，在调度器将任务状态标为 **running** 时，Thermos 开始运行进程。如果任务长时间处于 **pending** 或 **assigned** 状态，调度器会将其标为 **lost**，同时重新创建处于 **pending** 状态的新任务。任务正常结束后，会进入 **finished** 或 **failed** 状态，而强制终止会变为 **killing** 或 **restarting** 状态。

在资源需求过高，或硬件资源缩水（比如电池故障）时，一些重要的作业需要相比其他任务更高的优先级。Aurora 使用生产作业的概念来定义作业，它们比非生产作业更为重要，在需要释放资源时，非生产作业可能会被强行终止。这些被强行终止的任务会被标为 `PREEMPTING` 状态，之后最终会变为 `killed` 状态。

运行 Aurora

按照下列步骤在 Mesos 上安装 Aurora：

1. 安装 Mesos。
2. 本书撰写时，还没有 Aurora 的发行版，只能从源码构建。克隆 Aurora 的源代码：

```
ubuntu@master:~ $ git clone http://git-wip-us.apache.org/repos/asf/
aurora.git
ubuntu@master:~ $ cd aurora
```

3. Aurora 使用 Gradle 来做 build，这就是说，Gradle 脚本会下载所有的依赖包，然后创建发行版：

```
ubuntu@master:~/aurora $ ./gradlew distZip
```

4. 上一步会在目录 `dist/distribution` 下生成 `auroa-scheduler-*.zip` ZIP 文件。需要将其复制到所有 Mesos 节点上，并解压缩到 `/usr/local`。然后就可以使用 Aurora 了：

```
ubuntu@master:~/aurora $ sudo unzip dist/distributions/aurora-sche
duler-*.zip -d /usr/local
ubuntu@master:~/aurora $ sudo ln -nfs "$(ls -dt /usr/local/aurora-
scheduler-* | head -1)" /usr/local/aurora-scheduler
```

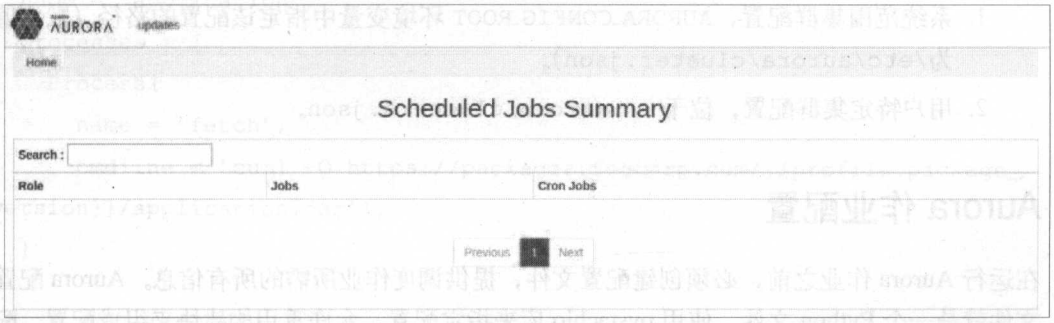
5. 指定合适的参数，启动 Aurora。运行 `-help` 参数会列出所有 Aurora 参数：

```
ubuntu@master:~/aurora $ /usr/local/aurora-scheduler/bin/aurora-
scheduler -help
```

6. 需要初始化 Aurora 使用的复制日志。
7. 本例中只使用了最少的配置来提供必需的参数。样例目录提供了复杂的样例配置：

```
ubuntu@master:~ $ /usr/local/aurora-scheduler/bin/aurora-scheduler
-cluster_name=test -mesos_master_address=zk://localhost:2181/mesos/
-serverset_path=/aurora/scheduler
```

上一步命令还会启动调度器，在 8081 端口启动 Web 接口：



注意，Aurora 提供了 vagrant 的搭建方式，可以很容易尝试 Aurora。如果想试试这种方式，需要 `cd` 到解压缩开的目录下，运行 `vagrant up` 命令。就会在一台虚拟机上安装 Mesos 和 Aurora。文档（`http://aurora.apache.org/documentation/latest/vagrant`）里有更多详细内容。

Aurora 集群配置

Aurora 集群配置描述 Aurora 集群，Aurora 客户端可以用短名来指向不同的集群。集群配置是 JSON 格式的，有如下字段：

- `name`：该配置里集群的名称，这是必需字段。
- `slave_root`：这是 Mesos slave 的工作目录。Aurora 客户端使用其定位 slave 上执行着的任务。这是必需字段。
- `slaverundirectory`：指定运行着的任务所处的目录名称。大部分情况下，该值必须为“latest”。这是必需字段。
- `zk`：这是 ZooKeeper 的主机名。

- zk_port：这是 ZooKeeper 运行的端口。默认值为 2181。
- schedulerzkpath：Aurora 服务器集注册到 ZooKeeper 上的路径。
- scheduler_uri：Aurora 调度器的 URI。如果指定了，就不会使用 ZooKeeper 的相关配置，而使用 scheduler_uri 直接连接调度器。因此，只在测试环境上推荐使用 scheduler_uri，生产环境中不推荐。
- proxy_url：如果指定了值，proxy_url 会被用来取代 master 的URI。
- auth_mechanism：这是客户端用来在 Aurora 调度器处完成认证的标识符。目前这部分工作还在进行中，只支持 UNAUTHENTICATED。

Aurora 客户端可以加载两个集群配置文件：

1. 系统范围集群配置，AURORA_CONFIG_ROOT 环境变量中指定该配置的路径（默认值为/etc/aurora/cluster.json）。
2. 用户特定集群配置，位于 ~/.aurora/clusters.json。

Aurora 作业配置

在运行 Aurora 作业之前，必须创建配置文件，提供调度作业所需的所有信息。Aurora 配置文件就是一个 Python 文件，使用 pystachio 库来指定配置。允许重用构建块来组成配置。配置文件必须有 .aurora 后缀。配置文件里有三种类型的对象：

- Process
- Task
- Job

配置文件顺序定义模板、进程、任务和作业。模板封装作业间的不同之处，这些地方无法用配置里的属性来定义。下述示例配置文件来自于 Aurora 文档：

```
# --- templates here ---
class Profile(Struct):
  package_version = Default(String, 'live')
  java_binary = Default(String, '/usr/lib/jvm/java-1.7.0-openjdk/bin/java')
  extra_jvm_options = Default(String, '')
  parent_environment = Default(String, 'prod')
  parent_serverset = Default(String, '/foocorp/service/bird/{{parent_environ
ment}}/bird')
```

```
# --- processes ---
main = Process(
  name = 'application',
  cmdline = '{{profile.java_binary}} -server -Xmx1792m '
            '{{profile.extra_jvm_options}} '
            '-jar application.jar '
            '-upstreamService {{profile.parent_serverset}}'
)
# --- tasks ---
base_task = SequentialTask(
  name = 'application',
  processes = [
    Process(
      name = 'fetch',
      cmdline = 'curl -O https://packages.foocorp.com/{{profile.package_
version}}/application.jar'),
  ]
)
# --- job template ---
job_template = Job(
  name = 'application',
  role = 'myteam',
  contact = 'myteam-team@foocorp.com',
  instances = 20,
  service = True,
  task = production_task
)
# -- job --
jobs = [
      job_template(cluster = 'cluster1', environment = 'prod')
                .bind(profile = PRODUCTION),
      job_template(cluster = 'cluster2', environment = 'prod')
                .bind(profile = PRODUCTION),
      job_template(cluster = 'cluster1', environment = 'staging',
                   service = False, task = staging_task, instances = 2)
```

```
        .bind(profile = STAGING),
]
```

Process 对象有两个关键属性：name 和 cmdline。name 属性必须是一个唯一的字符串，cmdline 是能被 bash shell 执行的脚本。与在 bash shell 上运行一样，可以组合多个 bash 命令。Process 对象可以指定如下可选属性：

属性	描述
max_failures	指定故障发生时重试的最大次数，之后该进程会被标为永久失败，并且不再重试。默认值是 1。该值设为 0 时，表示不限次数重试
daemon	在 min_duration 时间后，重新调用 Daemon 进程。当 max_failures=0 和 daemon=True 时，进程会无限次重试
ephemeral	设为 true 时，就会不用进程的退出状态来决定任务的完成情况。默认是 False
min_duration	重新运行相同任务之间的间隔时间，以秒计。默认值为 15。用来抵御对调度器的拒绝服务攻击
final	该进程是最终进程，必须在所有其他进程完成或失败之后运行。Thermos 在最终阶段调用最终进程。通常用于清理环境和统计。默认值为 False

Aurora 有很多 helper 对象和功能可以用来简化复杂任务的创建。一个 task 对象包括三个关键属性：

- name：任务的名称。如果没有指定，会被设为第一个进程的名称。
- processes：进程的无序列表。
- resource：决定执行任务所需的资源。resource 对象包含 CPU、RAM 和 disk 属性。

还支持如下可选属性：

- constraints：指定任务里进程间的依赖关系。目前，只支持 order 约束，指定运行进程的顺序。
- max_failures：指定多少进程失败后会将任务标为失败，默认值为 1。
- max_concurrency：指定 Thermos 在任务里能够并发运行的进程最大数目。默认值为 0，表示不限数目的并发。在执行消耗大的进程时该参数很有用。

- finalization_wait：控制执行清理的秒数。等待指定秒数后，会向进程发送SIGKILL 信号。

job 对象是可以运行在集群上的完全相同的任务的组。job 对象必须指定的属性为：

- task：指定该作业绑定到的任务。
- role：用来在 Mesos 上运行作业的用户账号。
- environment：指定作业必需提交到的环境。常用的值为 devel、test、prod 等。
- cluster：用来提交作业的 Aurora 集群的名称。可以是定义在 /etc/aurora/clusters.json 或者 ~/.clusters.json 里的任意值。

如下是 job 对象的可选参数：

属性	描述
instances	这是运行该作业的任务所必需的实例/副本数量，默认值为 1
priority	指定任务抢占优先级。高数值比低数值的优先级高，因此高数值作业可以抢占低数值作业的资源。默认值为 0
production	检查作业是否是生产作业。所有生产作业比非生产作业优先级高。默认值为 False
update_config	指定滚动更新策略
health_check_config	控制任务的健康检查
constraints	这是 Python 的 map 对象，指定作业的调度限制条件。默认值为空
contact	这是作业所有者的 email ID
max_task_failures	指定任务失败的最大次数，之后作业被标为失败。默认值为 1。-1 表示允许无限次失败
service	检查作业是否是服务作业。服务作业持续运行，并在执行成功或失败后重启。默认值为 False

下面是一个简单作业的完整配置文件，该作业输出 Hello world：

```
import os
hello_world_process = Process(name = 'hello_world', cmdline = 'echo Hello
world')
```

```
hello_world_task = Task(
  resources = Resources(cpu = 0.1, ram = 16 * MB, disk = 16 * MB),
  processes = [hello_world_process])

hello_world_job = Job(
  cluster = 'cluster1',
  role = os.getenv('USER'),
  task = hello_world_task)

jobs = [hello_world_job]
```

Aurora 客户端

一旦完成了配置文件，就可以使用 Aurora 客户端来启动作业并之交互了。Aurora 客户端命令根据操作分成很多子命令：

命令	子命令	描述
job	create	创建作业或服务
	list	列出匹配 jobkey 或者 jobkey 模式的作业
	inspect	分析作业配置文件，显示信息
	diff	将作业配置文件和运行着的作业比较
	status	给出已调度作业或作业组的状态信息
	kill 和 killall	强行终止已调度作业的实例
	update	开始某个运行着的作业的滚动更新
	cancel-update	取消进程的更新
	restart	启动作业实例的滚动重启
task	run	在运行作业实例的机器上执行给定 shell 命令
	ssh	在运行任务实例的机器上开启 SSH 会话
sla	get-task-up-count	打印日志范围内任务正常运行百分比。指定持续时间
	get-job-uptime	打印作业的正常运行时间百分比数值。指定想要达到的作业正常运行时间百分比数值

（续表）

命令	子命令	描述
quota	get	打印给定角色的配额
cron	start	立刻启动 cron 作业，不考虑其正常调度配置
	show	显示 cron 作业的调度状态
	schedule	为作业创建 cron 调度器。作业已有的调度器会被新建的取代
	deschedule	移除作业的 cron 调度器
config	list	列出配置文件里定义的所有作业
beta-update	start	在运行着的作业上启动调度器驱动的滚动升级
	status	显示调度器驱动更新的状态
	list	列出所有匹配查询的正在进行的调度器驱动作业
	pause	暂停调度器驱动更新
	resume	恢复调度器驱动更新
	abort	终止正在进行的调度器驱动更新

job 子命令提供了命令和 create 子命令交互，会创建并运行 Aurora 作业。jobkey 的格式为 <clust er>/<role>/<env>/<jobname>（比如，cluster1/dev-team/devel/ad-exp）。该命令会输出 job URL：

```
ubuntu@master:~ $ aurora create jobKey conf.aurora
```

本书撰写时，Aurora 项目还没有为 Aurora 提供 REST API，不过相关工作正在 Aurora 社区里进行。Aurora 命令行客户端很强大，有很多丰富的功能和各种命令。要注意，之前介绍的每个命令都有很多参数。Aurora 还有很多其他的有用功能，比如，允许改变或扩展 Aurora API 行为的 hook 点，性能监控及提供作业和集群详细信息的 Web UI。本书对这些 Aurora 特性不作深入介绍，详细信息可以参考 Aurora 文档（http://aurora.apache.org/documentation/latest）。

Aurora 样例

Aurora 项目在 examples 目录下有一些 Aurora 作业定义示例。尝试运行一个简单的 Aurora 作业，每 10 秒钟打印一次 **hello world**。如下是 example/jobs/hello_world.aurora 文件：

```
hello = Process(
  name = 'hello',
  cmdline = """
    while true; do
      echo hello world
      sleep 10
    done
  """)

task = SequentialTask(
  processes = [hello],
  resources = Resources(cpu = 1.0, ram = 128*MB, disk = 128*MB))

jobs = [Service(
  task = task, cluster = 'devcluster', role = 'www-data', environment =
'prod', name = 'hello')]
```

Aurora create 命令以 JobKey 和 .aurora 配置文件作为参数。注意，jobKey 的格式为<cluster>/<role>/<env>/<jobname>；在上面的例子里是 /devcluster/www-data/devel/hello_world。

如下命令会使用 hello_world.aurora 配置文件在 prod 环境里启动 hello 服务：

```
ubuntu@master:~ $ aurora create /devcluster/www-data/devel/hello_world
aurora/examples/jobs/hello_world.aurora
```

该命令启动 hello world 作业，打印出 URL，URL 里可以看到该 job 的详细信息。作业输出里看到 **hello world** 字样。

Aurora cron 作业

Aurora 也支持和 cron 类似的作业循环执行。Aurora 通过 job 对象里的 `cron_schedule` 属性识别出 cron 作业。该属性值是满足定时任务语法的受限集合，指定该作业的重复调度。`cron_collision_policy` 字段指定当之前运行的作业在达到下一个作业启动时间时还没有完成时的行为。`KILL_EXISTING` 会杀死旧实例，使用当前配置创建新实例。而 `CANCEL_NEW` 会取消新作业的运行。`KILL_EXISTING` 是默认策略。注意，Aurora 只会重试任何失败的 cron 作业 `max_task_failures` 次。将 `max_task_failuers` 设为 `-1`，可以让 Aurora 服务一直运行到发生故障为止。Aurora cron 文档（`http://aurora.apache.org/documentation/latest/cron-jobs`）详细介绍了如何用 Aurora 运行 cron 作业。

服务发现

当同时运行应用程序多个副本时，必须能够发现它们在哪里运行，并能够连接它们。这在出现故障时尤其重要，这时 Mesos 会在其他机器上重启服务，但是客户端需要能够连接上新启动的服务。发现服务的两种常用做法是：

- 基于域名服务
- 专业的服务发现方案（ZooKeeper、Consul、Serf、SkyDNS 等）

基于**域名服务**（**DNS**）的服务发现应用很广泛，因此，可以很轻松地和很多工具集成，而专业方案要求理解不同框架特定的 API。另一方面，使用基于 DNS 的发现机制，可能很难提供服务相关的丰富信息，但是专业方案能够提供除服务健康和复杂之外更多的信息。本书讲述一种基于 DNS 的服务发现机制—— Mesos-DNS，不过也有其他和 Mesos 集成良好的服务发现方案。Mesos-DNS 优于其他基于 DNS 服务发现框架的一点是不需要重写代码就可以天然利用 Mesos 的功能。另外，从 Mesos 0.22 版本开始，Mesos 在任务和执行器消息里包含了发现信息，这使得和服务发现工具的集成更加容易。

Mesos-DNS

Mesos-DNS（`http://mesosphere.github.io/mesos-dns`）是针对 Mesos 的基于 DNS 的服务发现机制。DNS 是互联网使用的层级分布式命名机制。比如，浏览器会使用 DNS 来发现特定网站的 IP 地址。从根本上而言，DNS 是设计用来从名称解析地址的。类似地，Mesos-DNS 将分配给 Mesos 应用程序的名称解析为这些服务运行的 IP 地址和端口。

本书撰写时，Mesos-DNS 刚发布 alpha 版本，仍然在进一步开发改进中。

Mesos-DNS 接口非常简单：它接受 DNS 请求，回复 DNS 记录。这样的设计使得 Mesos-DNS 的工作对于框架完全透明。框架只需使用标准的 DNS 查询就可以通过名称解析出地址。因此，当集群状态改变时，可以很容易地切换到其他机制来更新 DNS 记录，而无须改变框架本身。Mesos-DNS 有两个主要组件：

- DNS 记录生成器：记录生成器负责为所有运行着的应用程序生成 DNS A 和 SRV 记录。它周期性地查询 master，得到所有运行的应用程序及每个应用运行的任务的信息。因此，记录生成器知道所有任务的状态，比如启动、完成、失败或重启。记录生成器以 DNS 记录的格式保存各种服务的最新状态。
- DNS 解析器：解析器负责处理 DNS 请求并且回复请求。解析器直接回复 Mesos 上运行着的任务，为外部请求选择随机的 DNS 服务器，就像普通 DNS 服务器的行为一样。

目前，Mesos-DNS 只支持 ANY、A 和 SRV DNS 记录。对于 Mesos 域里的其他类型的记录，会返回 NXDOMAIN。也就是说它不能用于需要 PTR 记录的反向寻找。A 记录将主机名和 IP 地址关联起来。Mesos-DNS 为 `task.framework.domain` 主机名生成一个 A 记录，该主机名提供指定 slave 的 IP 地址，运行框架启动的任务。比如，其他 Mesos 任务可以通过 `db.marathon.mesos` 来为 Marathon 框架启动的 db 服务发现 IP 地址。另外注意，只会为 Mesos 分配端口的任务生成 SRV 记录。为防框架启动了多个同名任务，Mesos-DNS 会返回多个 DNS 记录，每个任务一条记录，记录顺序是随机的。这提供了基本的复杂均衡能力。

Mesos-DNS 没有提供其他功能，比如健康检查或者应用的生命周期管理。这有助于保持 Mesos-DNS 架构的简单和无状态。它不使用持久化存储、复制或一致性策略。无状态架构使得在大规模 Mesos 集群里非常易于扩展。该设计的后果之一就是 Mesos-DNS 本身无法容错，必须依赖于外部实体。比如，用 Marathon 启动 Mesos-DNS 或者 Aurora 框架来监控它，在发生故障时重启。Mesos-DNS 在 Mesos master 不可用时还能继续为现有服务提供 stale DNS 记录。注意，Mesos-DNS 能够处理 Mesos master 故障转移，因为它会连接到新的 Mesos master 上。

安装 Mesos-DNS

Mesos-DNS 用 Go 语言实现，可以被安装在能连接到 Mesos master 的任何机器上。按照下列步骤安装 Mesos-DNS：

1. 需要安装 Go 来编译源码。如下命令会安装 Go 并且设置合适的环境变量：

```
ubuntu@master:~ $ wget https://storage.googleapis.com/golang/go1.4.
linux-amd64.tar.gz
ubuntu@master:~ $ tar xzf go*
ubuntu@master:~ $ sudo mv go /usr/local/.
ubuntu@master:~ $ export PATH=$PATH:/usr/local/go/bin
ubuntu@master:~ $ export GOROOT=/usr/local/go
ubuntu@master:~ $ export PATH=$PATH:$GOROOT/bin
ubuntu@master:~ $ export GOPATH=$HOME/go
```

2. 安装 Mesos-DNS 及其依赖的 go 库：

```
ubuntu@master:~ $ go get github.com/miekg/dns
ubuntu@master:~ $ go get github.com/mesosphere/mesos-dns
ubuntu@master:~ $ cd $GOPATH/src/github.com/mesosphere/mesos-dns
ubuntu@master:~ $ go build -o mesos-dns main.go
ubuntu@master:~ $ sudo mkdir /usr/local/mesos-dns
ubuntu@master:~ $ sudo mv mesos-dns /usr/local/mesos-dns
```

Mesos-DNS 配置

Mesos-DNS 配置很简单，接受 JSON 配置文件为输入。默认来说，Mesos-DNS 会在当前目录查找 `config.json` 文件。还可以使用 `-config` 命令行参数指定配置文件路径。Mesos-DNS 有如下可选配置：

- `masters`：指定 Mesos master 的地址列表，以逗号分隔。
- `refreshSeconds`：指定 Mesos-DNS 根据来自 Mesos master 的信息更新 DNS 记录的频率。默认值是 60 秒。
- `ttl`：指定 Mesos-DNS 所给的 DNS 记录的有效时间。这是 DNS 记录里的标准 `ttl` 字段。`ttl` 值必须至少和 `refreshSeconds` 一样大。`ttl` 默认值是 60 秒。也就是说，一旦服务从 Mesos-DNS 处查询到服务地址，在再次查询之前，结果可以保留在缓存中 60 秒。因此，`ttl` 值越大，Mesos-DNS 的负载越小，也意味着当服务地址改变时，客户端会越晚得到这个信息。
- `domain`：这是 Mesos 集群的域名。域名可以是大小写字母、数字和“-”的组合，“-”可以作为域名的非首尾字母。“.”可以作为域名不同部分的分隔符。默认值是 `mesos`。

- port：指定 Mesos-DNS 侦听 DNS 请求的端口号。请求可以使用 TCP 或者 UDP 发送。默认值为 53。
- resolvers：逗号分隔的外部 DNS 服务器的 IP 地址列表。这些 DNS 服务器用来解析所有域外部的 DNS 请求。可以使用公开 DNS 服务器，比如 8.8.8.8（Google 提供），或者其他站点特定的 DNS 服务器。如果解析器没有设置正确，任务就无法解析任何外部名称。
- timeout：该值指定等待连接以及外部 DNS 解析器响应的时间。
- email：指定维护 Mesos 域的管理员的 email 地址。格式为“邮箱名称.域”。

如下是 build 目录下的配置文件示例：

```
ubuntu@master:~ $ cat /usr/local/mesos-dns/config.json
{
  "masters": ["zk://localhost:5050"],
  "refreshSeconds": 60,
  "ttl": 60,
  "domain": "mesos",
  "port": 53,
  "resolvers": ["169.254.169.254","10.0.0.1"],
  "timeout": 5,
  "resolvers": "10.101.160.16",
  "listener": "root.mesos-dns.mesos"
}
```

运行 Mesos-DNS

如下命令会使用当前目录的 config.json 在后台启动 mesos-dns。因为 mesos-dns 绑定到了特权端口 53，所以必须用 root 用户启动 mesos-dns：

```
ubuntu@master:~ $ sudo mesos-dns &
```

如之前所述，推荐从外部（使用 Marathon/Aurora/runit 等）启动 mesos-dns。一旦 mesos-dns 运行起来，可以使用 dig 来查询所有已有运行的服务来验证其工作正常。注意，任何配置上的改动都需要重启 mesos-dns 才能生效。mesos-dns 有 verbose 模式用来调试，使用 -v 参数启动 mesos-dns 就可以使用 verbose 模式。

Mesos 任务要想使用 Mesos-DNS 来解析 Mesos 上运行着的服务的地址，必须将 Mesos-DNS 设置为 slave 上的首选 DNS 服务器。需要在每个 slave 上将运行着的 mesos-dns 的主机地

址添加为命名服务器。比如，如果 mesos-dns 运行在 IP 地址为 10.2.3.4 和 10.2.3.5 的机器上，需要在每个节点上添加如下行到 /etc/resolve.conf 的开头，这样就可以确保 slave 上启动的任务会首先联系 mesos-dns 来解析服务地址。master 不需要任何修改。

```
ubuntu@master:~ $ cat /etc/resolv.conf
nameserver 10.2.3.4
nameserver 10.2.3.5
...
```

打包

现代应用程序代码不是独立运行的，而会依赖于其他应用程序和库函数。因此，部署时需要确保满足了这些依赖条件。Jar（Java 打包文件）、Python egg 和静态库都是打包的格式。有项目正在尝试将 Google 的 Kubernetes 项目（http://kubernetes.io）整合进 Mesos（https://github.com/mesosphere/kubernetes-mesos）里。Kubernetes 也允许大规模运行服务。Docker（https://www.docker.com）最近大放异彩，成为便携的打包应用程序的优先方式。本书第 6 章中会探讨 Docker 和 Mesos 的集成。

小结

本章讲述了 Marathon 和 Aurora 调度器框架，帮助在 Mesos 集群上长期运行服务。也探讨了如何在 Mesos 上使用 Chronos 循环启动作业。注意，Mesos 上还有一些本书没有涉及的其他框架可以用来部署服务，其中值得一提的是 HubSpot 开发的 singularity 框架（https://github.com/HubSpot/Singularity）。这些框架辅助 Mesos 向其将整个数据中心变成一台机器的目标一步步迈进。

本章还总结了Mesos 上的各种框架。探讨了解决不同业务问题的框架，但是并没有覆盖所有 Mesos 上的框架。在 https://github.com/dharmeshkakadia/awesome-mesos 上可以查看框架的完整列表。下章会钻研 Mesos 的工作机制。

第6章

理解 Mesos 内部机制

本章将介绍 Mesos 工作机制的内部细节。从 Mesos 的架构入手了解集群调度和公平性的概念，然后介绍 Mesos 中资源隔离及容错机制的实现。本章涵盖如下内容：

- Mesos 架构
- 资源分配
- 资源隔离
- 容错
- Mesos 扩展

Mesos 架构

在现代企业中，大量多样的商业需求催生了大量各异的应用程序，这些应用程序大多是部署在多台商用主机上的分布式应用。现在大多数企业会为每个应用都建立单独集群来为其提供相互隔离的运行环境。这种静态划分集群的方式下资源使用率非常低，并且每个应用都需要重复实现分布式基础服务。这些重复工作不仅十分消耗时间和精力，并且加剧了构建和维护分布式系统的困难性，对开发人员和运维人员都是一个巨大的挑战——开发人员需要使应用可以弹性扩展，并能处理在大规模部署情况下所不可避免的故障情况；运维人员则需要对这些各自部署在独立环境的所有应用进行管理和容量控制。

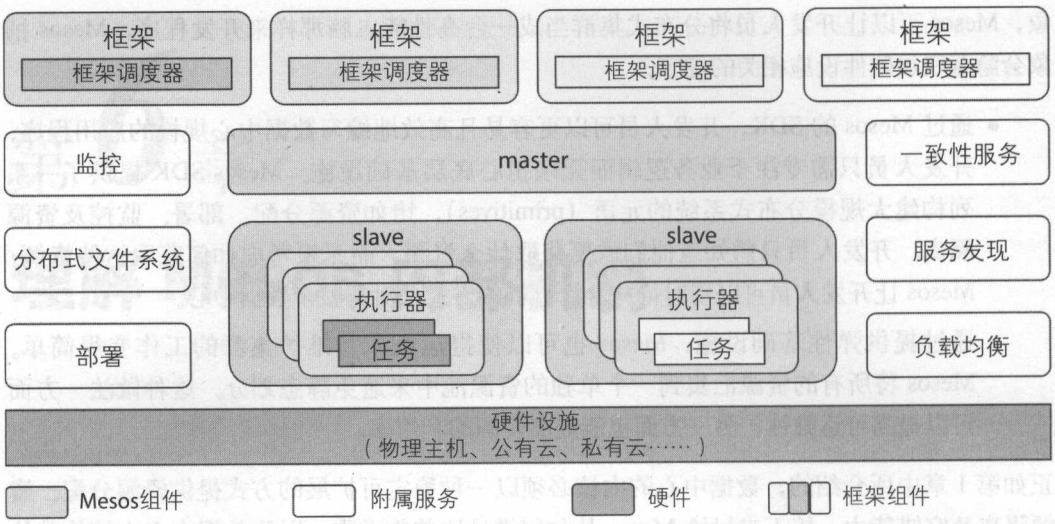
上述情况就好像是在没有操作系统的电脑上开发程序，需要自己去管理所有的硬件设备。Mesos 通过为数据中心提供一个操作系统内核来解决上述问题。通过提供一种高层次的抽

象，Mesos 可以让开发人员将分布式集群当成一台高性能电脑那样来开发程序，Mesos 抽象会隐藏底层硬件设施相关的细节。

- 通过 Mesos 的 SDK，开发人员可以更容易且高效地编写数据中心规模的应用程序。开发人员只需专注于业务逻辑而无须担心底层基础设施。Mesos SDK 提供了一系列构建大规模分布式系统的元语（primitives），诸如资源分配、部署、监控及资源隔离。开发人员只需知道他们需要获取什么资源，而无须考虑如何获取这些资源。Mesos 让开发人员可以用对待一台计算机的方式来对待整个数据中心。
- 通过提供弹性基础设施，Mesos 也可以使得运维人员维护集群的工作变得简单。Mesos 将所有的资源汇集到一个单独的资源池中来避免静态划分。这种做法一方面可以提高可运维性，另一方面也能提高资源的利用率。

正如第 1 章中所介绍的，数据中心的内核必须以一种稳定可扩展的方式提供资源分配、资源隔离及容错能力。接下来讨论 Mesos 是如何满足这些需求的，以及数据中心内核的其他重要概念。

- 伸缩性：内核需要具备根据机器和应用数量自动伸缩的能力。随着机器和应用数量的上升，调度的响应时间应该保持在可接受的范围内。
- 灵活性：内核需要能支持足够多的应用，可以兼容多种现有的集群框架，并能适应未来框架的发展。由于大部分集群的发展都需要一段时间，集群里会运行不同种类和型号的硬件，内核需要能很好地处理集群中的异构硬件。
- 可维护性：内核将作为现代基础设施中的重要一环，需要能够不断适应新需求。
- 动态资源利用：内核需要根据资源需求和现有硬件资源的条件不断进行调整以达到最佳的资源利用率。
- 公平性：内核需要公平对待不同用户和框架的资源请求。接下来的章节会具体介绍公平性的细节。



最小化接口是 Mesos 背后的设计哲学，这可以使得不同框架之间的资源共享更为高效，并将任务的真正调度和执行交给框架来负责，因此框架可以用多种方式来实现自己的调度和容错机制。这也使得 Mesos 的核心能够保持简洁，核心和框架可以独立并行发展。上图展示了 Mesos 的整体框架（`http://mesos.apache.org/documentation/latest/mesos-architecture`），它由如下几部分组成：

- Mesos master
- Mesos slave
- 框架
- 通信
- 附属服务

接下来将介绍每一部分所扮演的角色，以及 Mesos 作为数据中心的内核是如何满足不同需求的。

Mesos slave

Mesos slave 负责利用已有资源执行框架下发的任务。此外 slave 需要对运行中的任务进行合适的隔离，隔离机制还需要保证每个任务都能不多不少地获得所承诺的资源。

Mesos 所管理的 slave 上的资源可以通过 slave 资源和 slave 属性来进行描述。资源是 slave 任务运行中所要消耗的元素，而 slave 的其他相关信息则通过属性来描述。slave 资源由 Mesos master 统一管理并分配给不同的框架，属性则标识了每个 slave 的特殊信息，比如特

殊的操作系统、特殊的软件版本、特殊的网络环境或者当前 slave 的特殊硬件。这些属性都是简单的键值对，并随着分配给框架的资源 offer 进行传递。由于这些属性并不会被任务所消耗，因此在每次向框架提供 offer 时都会传递这些属性。Mesos 并不理解这些属性，由各个框架实现对这些属性的解析和利用。更多有关 Mesos 中资源和属性的信息可以参考：`https://mesos.apache.org/documentation/attributes-resources/`。

每个 Mesos 资源或者属性都可以归为如下几种类型。

- 标量（Scalar）：浮点数。
- 范围值（Range）：一组标量值，通过［最小值－最大值］的方式表示。
- 集合（Set）：任意的字符串。
- 文本值（Text）：属性中的任意字符串。

资源的名称由英文字母、数字、“-”、“/”、“.”、“_”组成。Mesos master 会特殊处理名称为 `cpus`、`mem`、`disk` 和 `ports` 的资源。没有 `cpus` 和 `mem` 资源的 slave 节点将不会为框架提供服务。这其中 `mem` 和 `disk` 通过以 MB 为单位的标量值表示，`ports` 以范围值来表示。slave 给不同框架提供的资源清单以 `--resources` 来表示，每个资源和属性之间以分号分割，例如：

```
--resources='cpus:30;mem:122880;disk:921600;ports:[21000-29000];bugs:{a,
b,c}'
--attributes='rack:rack-2;datacenter:europe;os:ubuntuv14.4'
```

该 slave 节点提供了 30 `cpus`、120GB `mem`、900GB `disk`、21000 到 29000 的 `ports`，以及 `a`、`b`、`c` 三个 `bugs` 资源。此外还有三个属性，`rack` 的值为 `rack-2`，`datacenter` 的值为 `europe`，`os` 的值为 `ubuntu14.4`。

Mesos 目前不提供直接支持 GPU 资源的方法，但是支持用户自定义资源类型。这就意味着如果我们指定 `gpu(*):8` 作为 `--resources` 的一部分，那么它就会成为提供给框架的资源 offer 中的一部分，然后框架就可以像使用其他资源一样来使用 GPU 资源。一旦有一部分 GPU 资源被任务所占用，其他任务就只能使用剩下的资源。与之类似，我们也可以给拥有 GPU 资源的 slave 节点一个新的属性，例如 `--attributes="hasGpu:true"`。Mesos 目前不支持 GPU 隔离，但是用户可以通过编写相关扩展来实现这个功能。

Mesos master

Mesos master 主要负责给各个不同的框架分配资源并管理任务的生命周期。Mesos 通过资源 offer 的形式来完成细粒度的资源分配。Mesos 作为资源的中间商可以利用各种可插拔的策略为框架提供资源，并根据相应策略将资源以 offer 的方式提供给框架。

在 Mesos 的世界中，资源 offer 代表了一个资源分配单位。它是一个由节点上可用资源组成的向量，代表着每个 slave 所提供给某个指定框架的资源。

框架

运行在 Mesos 之上的分布式应用称为框架。框架通过 Mesos 所提供的通用资源 API 来实现相关领域的需求。通常框架会运行许多任务，任务是资源的最终消费者，并且任务彼此之间可以不同。Mesos 中的框架由两部分组成：框架调度器及执行器。调度器负责协调任务的执行，执行器提供任务执行控制的功能。执行器可以通过多种方式来完成一个任务的执行，可以在一个执行器中运行一个任务，也可以在一个执行器中使用多进程来完成多个任务。除了生命周期和任务管理相关的功能外，框架 API 还提供了和调度器及执行器通信的功能。更多关于框架 API 的内容将会在第 7 章中介绍。

通信

Mesos 目前利用一个和 HTTP 类似的协议在各个组件之间进行通信。Mesos 利用 `libprocess` 库来实现通信，相关代码在 `3rdparty/libprocess` 中可以找到。`libprocess`库实现了进程之间的异步通信，它的通信原语和角色消息传递（actor message passing）的语义类似。`libprocess` 消息是不可变的，这也更易于内部消息的并发。下列 API 都使用了 Mesos 的通信机制。

- 调度器 API：框架调度器和 Mesos master 进行通信。该组内部通信目前仅由 SchedulerDriver API 使用。
- 执行器 API：执行器和 Mesos slave 之间进行通信。
- 内部 API：Mesos master 和 slave 之间进行通信。
- 运维 API：这组 API 主要是为运维人员准备的，并被 Web UI 等类似应用中使用。和其他大多数 Mesos API 不同，这是一组同步 API。

每个角色都需要通过 HTTP POST 请求来发送消息，消息的路径由角色名和消息名组成。为了和普通的 HTTP 请求区分开，请求头部的 `User-Agent` 域被设为“`libprocess/...`”。消息的数据存储在 HTTP 请求的 body 中。Mesos 使用 protocol buffers（`src/messages/messages.proto`）对消息进行序列化，消息的解析由消息的接收方实现。

下面是一个消息头部的例子，该消息通过在 `10.0.1.7:53523` 上的 `scheduler(1)` 向 `master` 注册一个框架：

```
POST /master/mesos.internal.RegisterFrameworkMessage HTTP/1.1 User-Agent:
```

```
libprocess/scheduler(1)@10.0.1.7:53523
```

Master 回复框架的消息头部如下所示：

```
POST /scheduler(1)/mesos.internal.FrameworkRegisteredMessage HTTP/1.1
User-Agent: libprocess/master@10.0.1.7:5050
```

在本书写作时，Mesos 社区正在初步讨论以纯 HTTP 方式重写调度器和执行器 API（https://issues.apache.org/jira/browse/MESOS-2288）。去除掉对 libmesos 的依赖后，使用标准的 API 将使得其他外部工具更容易同 Mesos 进行集成。同时在进行的工作还包括将内部消息转换成标准的 JSON 或者 protocol buffer 格式（https://issues.apache.org/jira/browse/MESOS-1127）。

附属服务

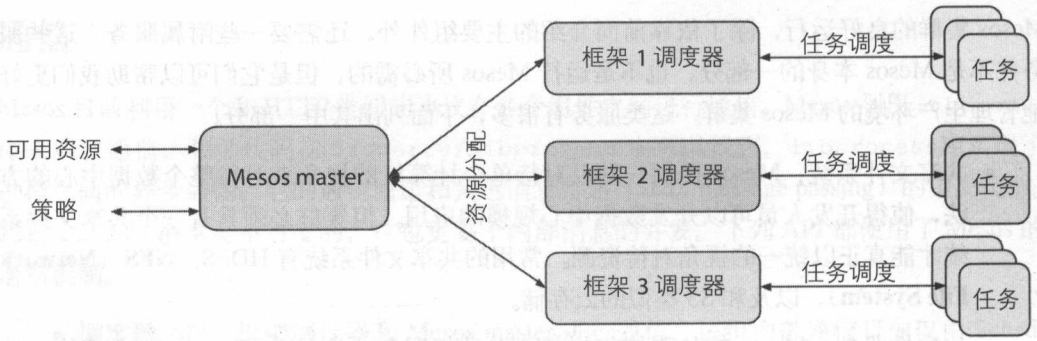
Mesos 集群的良好运行，除了依靠前面介绍的主要组件外，还需要一些附属服务。这些服务并不是 Mesos 本身的一部分，也不是运行 Mesos 所必需的，但是它们可以帮助我们更好地管理生产环境的 Mesos 集群。这类服务有很多，下面列出其中一部分。

- 共享文件系统：Mesos 提供一种以对待单个计算机的视角来对待整个数据中心的方法，使得开发人员可以开发数据中心规模的应用。但集群必须具有一个共享文件系统才能真正以统一的视角对待资源。常用的共享文件系统有 HDFS、**NFS**（**Network File System**），以及和 S3 类似的云存储。
- 一致性服务：Mesos 的故障恢复依赖于一致性服务。诸如ZooKeeper 和 etcd 的一致性服务在分布式环境中提供了可靠的选主服务。
- 服务编排（service fabric）：Mesos 使得用户能够以统一资源视角运行多个框架。随着越来越多的服务和应用的运行，如何让它们实现无缝连接变得越来越重要。例如，用户如何连接运行在 Mesos 上的 Hive 服务？Ruby on Rails 和 MongoDB 数据库两者之一或者均运行在 Mesos 上时，该如何相互发现并连接？网站的访问流量要如何路由到运行在 Mesos 上的网站服务器？上述问题可以通过服务发现和负载均衡机制解决，从而又需要 IP/port 管理和安全机制。我们将所有关于提供框架之间相互连接，以及框架和用户相互连接的服务统称为服务编排。
- 运维服务：运维服务是管理 Mesos 集群不可或缺的一部分。Mesos 的部署、升级、集群健康监控、报警、日志及安全，这些都是运维服务在集群管理中所发挥的不可替代的作用。我们将在第 8 章中介绍运维相关的细节。

资源分配

作为数据中心的内核，Mesos 需要为大量不同类型的负载提供服务，没有一个调度器可以满足所有这些框架的需求。例如，实时处理框架和长时间运行的服务相比调度方式是十分不同的，而批处理框架对资源的调度又和前两者都不一样。为了应对这个问题，Mesos 遵循了一个重要的设计原则：资源分配及任务调度的隔离。Mesos master 负责决定分配给每个框架多少资源，任务调度器负责决定如何使用这些资源，这取决于每个框架的调度器如何根据自身需求去实现。也可以这样理解 Mesos 的设计理念：Mesos 在各个框架间进行粗粒度的资源分配，每个框架根据自身任务的特点进行细粒度的任务调度。

Mesos master 从各个 slave 上获取可用资源的信息，并通过分配策略将资源分配给不同的框架。各个框架可以选择接受或者拒绝资源 offer。如果框架接受了资源 offer，就可以自由使用所获得的资源去执行任务。下图展示了 Mesos 资源分配的总体流程。



Mesos 双层调度器

下面是 Mesos 框架事件的一个典型流程。

1. 框架调度器在 Mesos master 中进行注册。
2. Mesos master 从 slave 获取资源 offer，调用分配模块的函数决定将这些资源分配给哪个框架。
3. 框架调度器从 Mesos master 接收资源 offer。
4. 当接收到资源 offer 后，框架调度器检查当前 offer 是否合适。如果合适，框架调度器接受该 offer，并向 master 返回一个需要在 slave 上利用这些资源运行的执行器列表。如果资源 offer 不合适，框架将会拒绝当前 offer，并等待更合适的 offer。
5. slave 分配所请求的资源并运行任务执行器。任务执行器在 slave 节点上运行框架下的任务。
6. 框架调度器接收任务运行成功或失败的通知。同时框架调度器继续接收资源 offer

和任务报告，并在合适的时机启动任务。

7. 框架从 Mesos master 注销，此后将不再接收任何资源 offer。需要注意的是这个操作并不是必需的，一些需要长时间运行的服务和元框架在正常的操作流程中并不会注销。

Mesos 的这种调度设计又被称为双层调度。由于资源分配的过程中不需要了解真正的调度何时运行，双层调度模型使得 Mesos 更加简单、易于扩展且更加稳定。框架和 Mesos 之间是松耦合的，可以各自独立迭代发展。并且框架的移植也会更加方便，下文会详细介绍。

双层调度的设计使得调度器并不了解全局的资源利用率，因此资源的分配可能并不是最优的。一种潜在的问题在于框架在执行任务时对资源有某些特殊需求，例如数据局部性、特殊的硬件、安全限制。在 Mesos 中，框架并不会将自己的这些特殊资源需求显式告知 Mesos master，相反框架会拒绝 master 发送的所有不符合需求的资源 offer。

Mesos 调度器

Mesos 是第一个可以在多个框架之间共享资源的集群调度器。Mesos 通过实时**支配资源公平算法（Dominant Resource Fairness，DRF）**进行资源分配。如果所有的资源都根据单一类型的资源进行静态划分，那么公平性很容易定义：将集群根据单一资源（如 CPU）平均分配给每个用户。DRF 将公平性的定义扩展到多种资源类型的组合，并且不需要进行静态划分。资源利用率和公平性这二者经常会相互冲突，但它们对于一个集群调度器来说都是十分重要的目标。确保所有用户/进程在数据中心这种共享环境的集群中获得数量接近的资源是资源分配的重要目标。

最大-最小资源公平算法是一个常用的在多个用户之间分配单一类型资源的机制，该算法对用户所能获得的最少资源进行最大化。在这种算法的最简单形式中，资源会被平均分给每一个人，加权最大-最小资源公平算法还支持优先级和资源预留。最大-最小资源公平算法也是许多其他操作系统和分布式框架的调度器的基础，例如 Hadoop 的公平调度器（`http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/FairScheduler.html`）、容量调度器（`https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html`）、昆西调度器（`http://dl.acm.org/citation.cfm?id=1629601`）等。但是这些调度器无法很好地处理 CPU、内存、磁盘和网络这样多类型资源组合的分配。多种资源组合的情况下需要重新定义公平性。例如，如果调度器收到 `<1 CPU, 3 GB>` 和 `<3 CPU, 1 GB>` 这两个资源请求时，该如何将两者进行比较，怎样分配才是公平的？

DRF 将最大-最小公平资源算法在多资源的情况下进行泛化。所占总资源比例最多的资源类型被称为该用户的支配资源。例如，总资源为 `<8 CPU, 5 GB>` 时，如果一个用户

申请<2 CPU, 1 GB>，那么支配资源为(2/8,1/5)中的最大值，也就意味着CPU为支配资源。一个用户的支配比例是指支配资源所占总资源的比例，在这个例子中是25%（2/8）。DRF在每个用户的支配资源上再利用最大-最小资源公平算法进行资源分配。DRF具有以下几个公认的特点。

1. 策略可验证性：用户无法通过夸大自己的需求获得更多的优势。
2. 鼓励资源共享：DRF算法可以保证每个用户所能获得的最少资源，所以相比独占总资源的1/N而言，用户更倾向于使用DRF分配算法。
3. 单一资源公平：在针对一种资源的情况下，DRF算法和最大-最小公平算法是等价的。
4. 无嫉妒（Envy-free）：每个用户都认为自己获得的资源比其他人的资源好（或至少不差）。这也意味着不同用户的相同请求都会得到相同的分配。
5. 瓶颈公平性：当一种资源成为瓶颈时，对于所有支配资源为该资源的用户，DRF算法和最大-最小公平算法等价。
6. 单调性：增加资源或者减少用户只会增加剩余用户所获得的资源。
7. 帕累托最优：DRF算法所产生的分配是帕累托最优的，意味着不可能改善某个用户的资源分配而不损害其他用户的利益。

本书不对DRF算法展开讨论，但非常推荐阅读DRF的论文来了解更多细节http://static.usenix.org/event/nsdi11/tech/full_papers/Ghodsi.pdf。

Mesos FrameworkInfo中的用户角色可以用来决定资源的分配。可以每一个用户或者每一个框架各自一个角色，也可以多个用户和框架共享一个角色。如果没有显式设置角色，Mesos会将角色设置为当前运行框架调度器的用户。

加权DRF

DRF会计算每个用户的支配资源比例，并将可用资源分配给拥有最小支配资源比例的用户。实际中，很少有机构会以完全公平的方式进行资源分配。大部分机构的资源分配都是有权重的，例如50%的资源给广告团队，30%的资源给测试团队，20%的资源给研发团队。为了满足这个需求，Mesos实现了加权DRF，master可以配置不同角色的权重。当权重被指定，客户端的DRF比例将会和对应权重相除。例如，权重为2的角色获得的资源将会是权重为1的角色获得资源的两倍。

Mesos中利用--weights和--roles启动参数来设置加权DRF，--weights后面接一个角色/权重的列表，格式为role1=weight1，权重值可以是非整数。

我们必须为在 `--roles` 启动参数中出现的每个角色都指定权重值。

资源预留

资源预留是另一个广泛存在的需求。例如，像 memcache 或者数据库这样在 Mesos 上运行的持久化或者有状态服务，需要一定的预留资源来保证服务不被机器重启所影响。如果没有资源预留，memcache 可能无法从 slave 中获得所需数据的资源 offer，这将会导致长时间的初始化，进而导致服务无法使用。资源预留同样可以限制每个角色所获得的资源。

资源预留可以保证每个角色所获得的资源，但是不恰当地使用将会导致资源的碎片化及集群资源的利用率低下。

所有的预留请求都需要通过 Mesos 的鉴权机制进行验证，以确保运维人员或者框架有足够的权限进行相关操作。资源预留权限和其他 ACL 配置都需要在 Mesos master 中进行设置。Mesos 支持如下两种预留方式：

- 静态预留
- 动态预留

静态预留

在静态预留中，资源为某个特定角色预留。如果该资源为角色 A 预留，那么只有用户 A 的框架可以使用这部分资源。静态预留通常由运维人员通过 `--resources` 选项来指定。该选项后面接一个针对每种资源的 `name(role):value` 的列表。没有指定角色的资源以及没有出现在 `--resources` 中的资源将会被划分到默认角色（*）名下。例如，`--resources="cpus:4;mem:2048;cpus(ads):8;mem(ads):4096"` 将拥有 8 个 CPU 4096MB 内存的 slave 预留给 “ads” 角色，4 个 CPU 和 2048MB 内存未做预留。如果想要改变静态预留，则需要删除 slave 所有的检查点后重启 slave。

如果各个 slave 的静态划分不一致，集群的管理很快就将变得十分复杂。

动态预留

动态预留允许运维人员和框架更加灵活地管理预留资源。框架可以利用动态预留来将 offer

预留，使得这部分资源之后只能被分配给同一个框架。

撰写本节时，动态预留机制正在被快速开发，并将会出现在下一个 Mesos 版本中（https://issues.apache.org/jira/browse/MESOS-2018）。

当申请预留资源时，Mesos 会尝试将非预留资源转换成预留资源。同样当归还预留资源时，会将之前预留资源归还到非预留资源池中。

为了支持动态预留机制，Mesos 允许在接受资源 offer 时返回一组 `Offer::Operations` 作为请求的响应。框架在接受资源 offer 时发送 `Offer::Operations::Reserve` 和 `Offer::Opeartions::Unreserve` 来管理预留资源。例如，一个框架接收了一个 32 个 CPU 和 65536MB 内存的资源 offer：

```
{
  "id" : <offer_id>,
  "framework_id" : <framework_id>,
  "slave_id" : <slave_id>,
  "hostname" : <hostname>,
  "resources" : [
    {
      "name" : "cpus",
      "type" : "SCALAR",
      "scalar" : { "value" : 32 },
      "role" : "*",
    },
    {
      "name" : "mem",
      "type" : "SCALAR",
      "scalar" : { "value" : 65536 },
      "role" : "*",
    }
  ]
}
```

框架可以通过发送 `Operation::Reserve` 消息在 `resources` 域指定所需资源状态来预留 8 个 CPU 和 4096MB 内存：

```
[
  {
```

```
    "type" : Offer::Operation::RESERVE,
    "resources" : [
      {
        "name" : "cpus",
        "type" : "SCALAR",
        "scalar" : { "value" : 8 },
        "role" : <framework_role>,
        "reservation" : {
          "framework_id" : <framework_id>,
          "principal" : <framework_principal>
        }
      }
      {
        "name" : "mem",
        "type" : "SCALAR",
        "scalar" : { "value" : 4096 },
        "role" : <framework_role>,
        "reservation" : {
          "framework_id" : <framework_id>,
          "principal" : <framework_principal>
        }
      }
    ]
  }
]
```

执行成功后，框架将会接收预留的资源 offer，下一个从 slave 发送的资源 offer 可能是这样：

```
{
  "id" : <offer_id>,
  "framework_id" : <framework_id>,
  "slave_id" : <slave_id>,
  "hostname" : <hostname>,
  "resources" : [
    {
      "name" : "cpus",
```

```
      "type" : "SCALAR",
      "scalar" : { "value" : 8 },
      "role" : <framework_role>,
      "reservation" : {
        "framework_id" : <framework_id>,
        "principal" : <framework_principal>
      }
    },
    {
      "name" : "mem",
      "type" : "SCALAR",
      "scalar" : { "value" : 4096 },
      "role" : <framework_role>,
      "reservation" : {
        "framework_id" : <framework_id>,
        "principal" : <framework_principal>
      }
    },
    {
      "name" : "cpus",
      "type" : "SCALAR",
      "scalar" : { "value" : 24 },
      "role" : "*",
    },
    {
      "name" : "mem",
      "type" : "SCALAR",
      "scalar" : { "value" : 61440 },
      "role" : "*",
    }
  ]
}
```

如上所示，框架将资源 offer 中的 8 个 CPU 和 4096MB 内存作为预留，24 个 CPU 和 61440MB 内存未作为预留。解除预留的做法和预留操作相反，框架在接收到资源 offer 时可以发送解除预留的消息，之后的资源 offer 将不会再有预留资源。

运维人员可以通过 /reserve 和 /unreserve HTTP 接入点对资源预留进行管理。运维 API 允许运维人员在不重新启动 slave 的情况下更改预留设置。例如，如下命令通过运维认证 principal ops 将 slave1 上的 4 个 CPU 和 4096 MB 内存作为 role1 的预留资源：

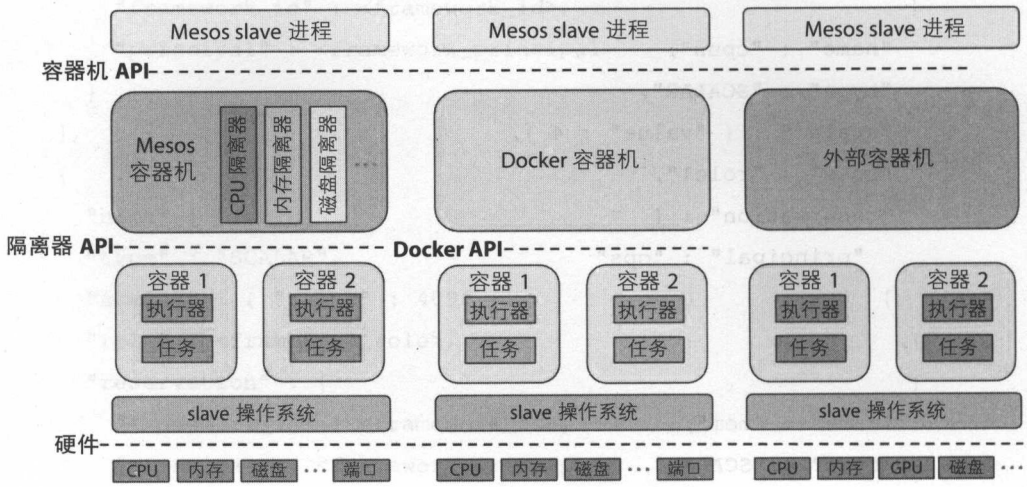
```
ubuntu@master:~ $ curl -d slaveId=slave1 -d resources="[
        {
          "name" : "cpus",
          "type" : "SCALAR",
          "scalar" : { "value" : 4 },
          "role" : "role1",
          "reservation" : {
            "principal" : "ops"
          }
        },
        {
          "name" : "mem",
          "type" : "SCALAR",
          "scalar" : { "value" : 4096 },
          "role" : "role1",
          "reservation" : {
            "principal" : "ops"
          }
        },
      ]"
      -X POST http://master:5050/master/reserve
```

在结束资源分配的讨论前，需要注意到 Mesos 社区还在不断地将有趣的想法和创新引入 Mesos 的资源分配中，例如学术论文和其他系统中所提到的超额认购（https://issues.apache.org/jira/browse/MESOS-354）。

资源隔离

Mesos 在 slave 上提供了多种隔离机制，以便为不同的任务提供沙箱环境。为某个框架任务或者用户分配资源不能影响正在运行的任务。容器作为一种轻量级虚拟机提供了必要的隔离机制，又不像虚拟机那样有大量额外的性能开销。利用容器我们可以限制每个进程及其子进程可以获得的资源。容器具体工作的细节不在本书的讨论范围内，为了了解 Mesos 的资源隔离机制，只需知道容器提供了一组资源隔离的功能即可。

Mesos 采用了一种插件的方式来设计资源隔离机制，并在项目诞生后不断发展。Mesos slave 利用容器机来为执行器和任务提供隔离的运行环境，下图展示了 slave 所使用的几种不同隔离机制。



Mesos slave 的不同隔离机制

容器机 API 的目的是支持更多不同的容器机实现，这意味着我们可以提供自己的容器机和隔离器。当 slave 进程启动后，我们可以指定使用的容器机和隔离器类型，来对资源进行限制。容器机的接口文件中包含了目前版本的容器机 API 的详细细节，可以在 `src/slave/containerizer/containerizer.hpp` 找到相关容器机 API 的定义。

Mesos 容器机

Mesos 容器机是一个内部容器机实现（`http://mesos.apache.org/documentation/latest/mesos-containerizer`），它提供了两种类型的隔离器：基于 POSIX 系统的进程级别的基础隔离，以及基于 Linux 内核特性 cgroups（也被称为 control group）的 cgroups 隔离。基于 cgroups 的隔离提供了更好的资源隔离、针对 CPU 和内存的统计，以及 checkpoint 等高级特性。隔离器 API 使我们可以为 slave 选择隔离机制进行资源隔离。需要注意的是，只有 `MesosContainerizer` 类型容器必须使用隔离器 API，外部容器机可以使用自己的隔离机制。这种组合设计十分灵活，运维人员可以选择使用多种不同的隔离器。除了基于 cgroups 的 CPU 和内存隔离，`MesosContainerizer` 还提供了磁盘隔离、共享文件系统隔离及 PID namespace 隔离。

在最初版本的Mesos中就已经支持磁盘空间隔离，Mesos允许对磁盘资源进行调度。从0.23版本开始，Mesos运维人员可以通过隔离器制定磁盘配额，周期性地检查任务磁盘使用空间，并杀掉那些超出使用配额的进程。该功能可以通过在salve上使用`--isolation`选项并指定`posix/disk`及`--enforce_container_disk_quota`选项来开启。默认情况下磁盘配额功能是关闭的，用户不会因为使用超出限制容量的磁盘空间而被杀掉。POSIX磁盘隔离器通过周期性地执行`du`命令并把磁盘使用率记录到`/monitor/statistics.json`来进行统计。周期间隔可以通过在slave上设置`--container_disk_watch_interval`选项来指定。默认的时间间隔为15秒。

在撰写本章内容时，Mesos对持久化存储的支持还在不断地改进中，读者可以通过`https://issues.apache.org/jira/browse/MESOS-1554`来跟踪项目的进度。

共享文件系统隔离器可以修改每个容器的存储系统视图。可以在框架的`ExecutorInfo`域的`ContainerInfo`中指定要进行的修改，也可以通过slave的`--default_container_info`选项来指定。`ContainerInfo`选项将指定的共享文件系统中的数据卷（`host_path`）映射到容器文件系统中（`container_path`）。数据卷在挂载时可以指定只读权限或者可读写权限。如果`host_path`是绝对路径，那么共享文件系统中所有以`host_path`为根目录的子目录，都可以在容器中通过`container_path`来进行访问。

如果`host_path`是相对路径，那么可认为它被挂载到相对执行器工作路径的目录位置上。该目录将会被创建并和共享文件系统中的目录拥有相同的权限位。一种常见的共享文件系统隔离器的使用场景是使得共享文件系统中的部分文件成为每个容器的私有文件。例如，通过`host_path="tmp"`和`container_path="/tmp"`，每个容器内都将有一个`/tmp`私有目录。这将会在执行器的工作目录中创建权限为1777的`tmp`目录，并将其挂载到container的`/tmp`下。这个步骤对于容器中的进程是透明的，容器内部将无法看到宿主机上的`/tmp`和其他容器的`/tmp`。

PID namespace隔离器可以给每个容器一个相互隔离的PID名字空间。这可以限制容器内部所能发现的进程，某个容器内的进程将无法和其他容器内的进程进行通信。由于某个PID namespace中的1号进程的结束不会杀死其他空间内的进程，所以进程结束的不干净问题也自然解决了。

Docker容器机

Docker（`https://www.docker.com`）是一个构建、分发和运行应用的开放平台。通过它可以轻松地将应用进行组合，并利用上层API以可移植的方式运行轻量级的Docker容器。Docker容器可以利用cgroups、LXC、OpenVZ和内核级别的namespaces进行隔离。作

为一种应用打包的方式，Docker 越来越流行。由于 Docker 提供了一种将应用打包至容器的方式，而 Mesos 提供了大规模运行容器的方式，所以 Mesos 和 Docker 的联系越来越紧密。从 Mesos 0.20 版本开始，Docker 就成为了 Mesos 世界的一等公民（http://mesos.apache.org/documentation/latest/docker-containerizer），并且有着原生的内建实现，使用 Docker 无须任何额外的外部依赖。

Mesos 0.19.0 以外部容器机的形式支持 Docker。在这种模式下 Mesos 并不知道 Docker 的细节情况，所有 Docker 相关的特性都需要其子进程自己控制。

DockerContainerizer 主要工作是将任务或者执行器的启动和销毁过程翻译成对应的 Docker CLI 命令。DockerContainerizer 解析 docker run 的参数，因此未来 Mesos 可以透明地过渡到利用 Docker remote API。下面是使用 Docker 容器机的步骤。

1. 执行器将所有在 CommandInfo 中指定的文件放入沙箱。
2. 执行 Docker pull 命令获取镜像。

这一步根据镜像大小和网络状况可能会花费不同的时间。如果时间超出了 executor_registration_timeout 值（默认为 1 分钟），容器机会失败。这时需要将 executor_registration_timeout 值调大。

从 Mesos 0.20.1 版本开始，只有在镜像包含 tag :latest 时才会运行 Docker pull 命令，而不是在每次容器启动时都去 pull 镜像。这样可以避免一部分因为无法连接到远端镜像仓库而导致的任务启动失败现象。

3. 利用 DockerExecutor 运行 Docker 镜像。这一步会将沙箱目录映射到 Docker 容器，并且设置 MESOS_SANDBOX 环境变量为目录映射关系。
4. 将 Docker 运行日志输出到沙箱的 stdout 和 stderr 文件。
5. 在容器机退出或者销毁时，停止并删除所有的 Docker 容器。

为了使用 Docker，需要在 slave 上安装 Docker CLI 客户端（版本要求最低为 1.0.0），并且在 Mesos 的启动参数 --containerizers 中指定 Docker。Docker 容器机允许通过 containerInfo.DockerInfo.image 选项指定需要运行哪个 Docker 镜像。.dockercfg 文件中包含登录私有仓库的信息，我们可以通过 URI 来指定这个文件将.dockercfg 文件放入沙箱环境中。由于 Docker 容器将 HOME 环境变量指向了沙箱，Docker CLI 可以自动使用所提供的配置文件。我们也可以通过 TaskInfo 和 ExecutorInfo 中的 ContainerInfo 域对 Docker 进行设置。

直到 Mesos 0.20 版本，TaskInfo 只包含 CommandInfo（通过 bash 命令来执行任务）或者 ExecutorInfo（启动一个用户自定义执行器来运行任务）。

镜像是容器快照所保存成的一个不可修改的文件。用户可以选择将 Docker 镜像以任务或者执行器的方式启动。

以任务形式运行 Docker 镜像需要设置 TaskInfo 域中的 command 和 container。Docker 容器机将会利用这个命令来启动 Docker 镜像。ContainerInfo 需要被设置为 docker，并且在 DockerInfo 中需要指定所要运行的 Docker 镜像。

以执行器的方式来运行 Docker 镜像需要在 ExecutorInfo 中设置 ContainerInfo。ContainerInfo 中 type 必须被设置为 docker，CommandInfo 将被用来启动执行器。在这种场景下，Docker 镜像将被当作 Mesos 执行器来启动，并会在启动后同 slave 一起注册。同时，以这种方式启动 Docker 镜像时，Mesos 并不会启动命令执行器，而是使用 Docker 容器的 PID 为执行器。

Docker 镜像目前支持一个 entrypoint 和一个可选的 CMD 命令。运行一个设置了 entrypoint 的镜像时，CommandInfo 中的 shell 选项必须被设置为 false。如果 shell 选项为 true，DockerContainerizer 将会把用户命令包含在 /bin/sh -c 中，并当作镜像 entrypoint 的一个参数。

如果希望运行 Docker 的默认 CMD 命令（docker run image），则不能设置 CommandInfo 域。否则该命令会覆盖默认命令。

DockerContainerizer 启动所有名字形式为“mesos-”前缀加 SlaveId 的容器，并假定所有容器都可以通过这个名字来停止和删除。Docker 容器机如果开启了 docker_mesos_image 选项，则可以在 slave 重启后恢复容器执行。Docker 容器机自己也是在容器中运行的，因此它也可以使用相同的恢复机制进行恢复。目前运行 Docker 镜像的默认网络模式是 host 模式。从 0.20.1 版本开始，Mesos 开始支持桥接网络模式。

外部容器机

对于外部容器机，容器中执行器的隔离需要由外部容器机自己去实现和管理（http://mesos.apache.org/documentation/latest/external-containerizer）。外部容器机的实现由两部分组成：**外部容器机（External containerizer，EC）**提供了基于 Mesos slave 外部插件的容器 API；**外部容器机程序（External Containerizer Program，ECP）**作为可执行的外部插件，实现了容器化的真正逻辑。

EC 通过 shell 进程传递命令参数来调用 ECP，其余数据通过 stdin 和 stdout 来进行传递，输入和输出信息采用了 protocol buffer 协议。containerizer::XXX 消息格式在 include/mesos/containerizer/containerizer.proto 中进行了定义，mesos::XXX 消息格式在 include/mesos/mesos.proto 进行了定义。

下面是 ECP 必须要通过命令形式来实现的命令：

launch < containerizer::Launch

launch 命令用来启动一个执行器。Launch 消息中包含了通过执行器启动任务所需的所有信息，这个命令不需要等待执行器或者命令返回。

update < containerizer::Update

update 命令用来更新容器资源限制，该命令从 Update 消息中获得 ContainerID 和 Resources，并且不产生任何输出。

usage < containerizer::Usage > mesos::ResourceStatistics

usage 命令用来获取当前容器的资源使用情况。该命令利用 Usage 消息中包含的 ContainerID 信息返回 ResourceStatistics 协议消息。ResourceStatistics 中包含了时间戳、CPU 及内存等资源的使用情况。

wait < containerizer::Wait > containerizer::Termination

wait 命令在执行器或者命令结束前一直处于阻塞状态。它接受 wait protocol buffer 消息并生成 Termination protocol buffer 消息。wait 消息中包含了 ContainerID，Termination 消息中包含了退出状态，布尔标识位 killed 记录了容器是否由于资源限制而被杀掉，字符串 message 域记录了结束的消息信息。

destroy < containerizer::Destroy

destroy 命令终止输入消息中 ContainerID 所对应的执行器。它不返回任何输出结果。

containers > containerizer::Containers

containers 命令返回所有当前正在运行的容器，并且不需要任何输入。

recover

recover命令允许 ECP 恢复执行器的状态。ECP 可以设置检查点并利用 recover 命令从检查点中恢复。

当启动一个任务时，EC 保证 ECP 在接收其他容器命令前首先收到一个启动命令。所有其他命令在队列中等待 ECP 响应启动命令。除了这个命令之外的其他命令可以以任意顺序执行。ECP 中的日志会被重定向到沙箱的日志文件中。ECP 的返回状态码标识启动是否成功，非零的状态码表明有错误发生。Mesos 容器机的文档中有详细的时序图解释 EC 和 ECP 环境下的容器生命周期。

所有容器机都可以从框架中的 `TaskInfo` 接收额外信息来配置将要启动的容器。例如，可以通过 `containerInfo.image` 来指定外部容器使用的镜像。ECP 还可以使用如下环境变量。

- `MESOS_LIBEXEC_DIRECTORY`：mesos-executor 和 mesos-usage 及其他一些文件的位置。
- `MESOS_WORK_DIRECTORY`：指定 slave 的工作目录，并用来区分不同的 slave 实例。
- `MESOS_DEFAULT_CONTAINER_IMAGE`：该变量只对启动命令有效，用来指定启动所使用的默认镜像。

为了在 Mesos 中使用外部容器机，slave 需要设置 `--isolation=external` 以及 `--containerizer_path=/path/to/external/containterizer` 选项。

容错

容错是数据中心操作系统中十分重要的需求。在发生故障的情况下仍然保持功能的正常运行是大规模系统必不可少的能力。Mesos 在架构中不存在单点，并且能够在多个实例出错的情况下继续正常工作。Mesos 有三种错误情况需要考虑：机器宕机、Mesos 进程 bug 和升级。我们主要需要在 Mesos master、Mesos slave 和框架这三个主要组件上考虑以上问题。

当 Mesos slave 主机宕机时，master 会发现宕机的情况，并给框架发送一个该 slave 宕机的事件。框架可以将当前运行在该 slave 上的任务重新调度到其他健康的 slave 上。当机器被修复，并且 slave 进程重新启动恢复到健康状态时，slave 会重新向 master 注册并再次成为 Mesos 集群中的一员。在发生 slave 进程错误或 slave 升级时，slave 进程可能在一段时间内不可用，但是在 slave 上的执行器不会受到影响。slave 进程在重启后会**恢复**这些任务，这一步骤也被称为 slave 恢复。

当框架调度器由于机器宕机、程序错误或者升级导致错误时，框架下的已有任务将会继续执行。如果框架实现了故障恢复，它会重新向 master 注册并获取所有任务的状态信息。

Mesos 利用选主来保证 master 的高可用。这意味着当 master 由于宕机、进程故障或者升级而导致 master 不可用时，另一个守护的 master 会被选举为主，所有的 slave 和框架，包括执行器及任务都会继续运行。slave 和框架会重新向 master 注册，因此 master 发生的错误

不会影响到整个集群的正常运行。Mesos 目前只支持 ZooKeeper 这一种选举服务，社区内部正在考虑加入诸如 etcd 等其他选举服务。下一小节将介绍如何使用 ZooKeeper 来保证 master 的高可用。

ZooKeeper

分布式协同服务是构建分布式系统时必不可少的一环，Apache ZooKeeper（`http://zookeeper.apache.org`）正是分布式协同服务的一种实现，它包含了一个名为 **ZooKeeper 原子性广播（ZooKeeper Atomic Broadcast，ZAB）**的一致性算法。ZAB 是一个高性能的广播算法，用来在主备系统中提供强一致性保证（`http://dl.acm.org/citation.cfm?id=2056409`）。ZooKeeper 经过了生产环境的考验，许多项目都依赖 ZooKeeper 来保证高可用。

为了使 Mesos 在高可用模式下运行，需要安装并运行一个 ZooKeeper 集群。一个 `N` 个节点的 ZooKeeper 集群可以在 `ceil(N/2)` 个节点失效的情况下仍然保持正常工作。我们将在 `zoo1`、`zoo2` 和 `zoo3` 这三个节点上安装 ZooKeeper，下面是具体的安装步骤。

1. 在 Mesos 的 `3rdparty` 文件夹中包含了 ZooKeeper 的发行版。也可以从 `http://zookeeper.apache.org` 获取上游的发行版：

   ```
   ubuntu@master:~ $ cd 3rdparty
   ```

2. 解压并进入目录：

   ```
   ubuntu@master:~ $ tar {xzf zookeeper-.tar.gz
   ubuntu@master:~ $ cd zookeeper-
   ```

3. 在 `conf` 目录下创建 ZooKeeper 的配置文件 `zoo.cfg`。可以利用发行版的默认样例配置，其中设置了 `tickTime=2000`, `dataDir=/tmp/zookeeper`, `clientPort=2181`, `syncLimit=5`, `initLimit=10`。

   ```
   ubuntu@master:~ $ cp conf/zoo_sample.cfg conf/zoo.cfg
   ```

   现在可以把 ZooKeeper 集群中的所有节点加入配置文件中，实例如下：

   server.1=zoo1:2888:3888
   server.2=zoo2:2888:3888
   server.3=zoo3:2888:3888

4. 启动时 ZooKeeper 的每个节点会根据 `conf` 目录下的 `myid` 来确定自己的身份。每个节点的 myid 都必须是 1~255 中唯一的一个数字。

5. 现在就可以启动 ZooKeeper 了：

```
ubuntu@master:~ $ bin/zkServer.sh start
```

可以设置 Mesos 的启动模式，使它使用 ZooKeeper 来保证高可用。

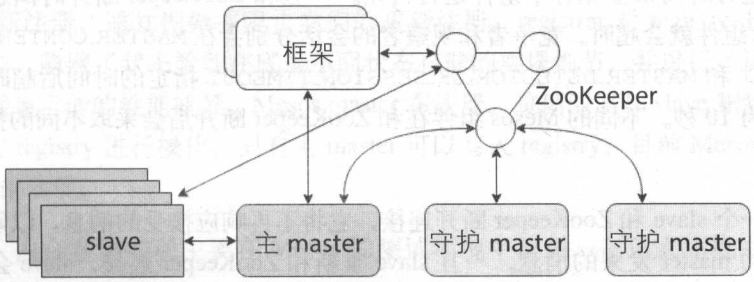
1. Mesos master 启动时添加 `--zk` 选项，需要指明所使用的 ZooKeeper 地址，以及 Mesos 所使用的名字空间（示例中使用 `/mesos` 作为路径）：

```
ubuntu@master:~ $ mesos-master --zk=zk://zoo1:2181,zoo2:2181,zoo3:
2181/mesos
```

2. Mesos slave 启动时添加 `--master` 选项，指向 ZooKeeper 的 URI 地址：

```
ubuntu@master:~ $ mesos-slave.sh --master=zk://zoo1:2181,zoo2:2181,
zoo3:2181/mesos
```

现在 Mesos slave 可以通过 ZooKeeper 来确定当前的 master。当出现失败时，ZooKeeper 会选举出一个新的 master，并将指向的新 master 告知 slave。下图描述了一个主 master 和两个守护 master 的部署情况。在高可用部署模式下，所有的 slave 和框架都通过 ZooKeeper 来获知当前 Mesos master 的地址。

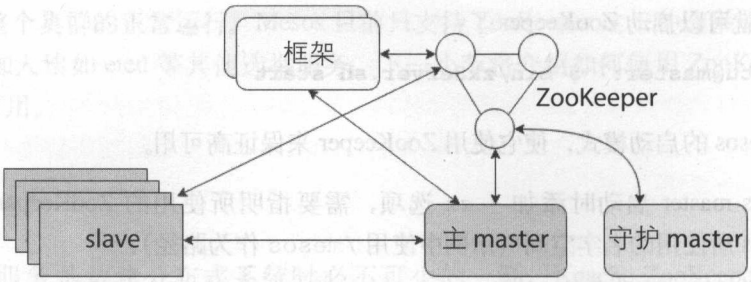


Mesos 在高可用模式下运行

如果当前的 Mesos master 发生故障，ZooKeeper 会从守护 master 中选举出一个新的 master，所有的 slave 和框架都将和新的 master 进行通信，如下图所示。

故障检测及处理

在 ZooKeeper 的选举模型中有竞争者和观察者两个抽象模型。竞争者试图成为 master，观察者想知道当前的 master 是谁。每个 Mesos master 既是竞争者也是观察者——一方面想要知道当前的 master 是谁，另一方面想要选举自己为新的 master。Mesos slave 和框架都是观察者，寻找当前的 master 并与之进行通信。ZooKeeper 的群组领导候选者通过追踪下列



Mesos 通过守护 master 进行故障恢复

ZooKeeper 会话事件来对注册和注销进行处理，同时也利用这些事件来对 Mesos 组件进行监控。

- 连接
- 重连
- 会话失效
- ZNode 创建、删除和更新

Mesos 利用超时时间来检测各个组件是否可用。一旦和 ZooKeeper 断开时间超过了配置的时间，Mesos 组件就会超时。竞争者和观察者的会话分别会在 `MASTER_CONTENDER_ZK_SESSION_TIMEOUT` 和 `MASTER_DETECTOR_ZK_SESSION_TIMEOUT` 指定的时间后超时，这两个时间默认被设为 10 秒。不同的 Mesos 组件在和 ZooKeeper 断开后会采取不同的措施，分别介绍如下。

- 如果一个 slave 和 ZooKeeper 断开连接，它将不再响应接受的消息，以确保不会执行从旧的 master 发来的消息。一旦 slave 重新和 ZooKeeper 连接，slave 会继续执行来自 master 的消息，此时的 master 既可能是新选举出来的也有可能是之前的 master。
- 如果一个调度器驱动和 ZooKeeper 断开连接，它将会通知调度器，并由调度器决定下一步应该如何处理。
- 如果 Mesos master 和 ZooKeeper 断开连接，将会进入无主状态。
  - 如果当前 master 是主，则它自己终止。需要管理人员来手动启动并将其以守护 master 的形式运行。
  - 如果当前 master 是守护模式，则它只需要等待和 ZooKeeper 重新连接。
- 如果由于网络隔离而导致 slave 无法和当前的主 master 进行通信，那么 slave 的健康检查会失败，master 会将 slave 设置为未激活状态。未激活的 slave 将不会向 master 重新注册，并且会被之后的消息要求主动关闭。该 slave 上的所有任务状态都会被标

记为 LOST，框架会对这些任务进行后续处理（例如重新运行）。

欲了解关于 Mesos 和 ZooKeeper 使用的更多细节，可以参考 `src/zookeeper/` 下的 `LeaderContender` 和 `LeaderDetector` 类的实现，它们实现了一个通用的 ZooKeeper 选主算法。`src/zookeeper/` 下的 Group 类实现了 ZooKeeper 的群组领导。本书将在第 8 章中介绍更多 ZooKeeper 配置使用方面的内容。

Registry

Mesos 的设计初衷是保持 master 的无状态，尽可能简单且可扩展。registry 使得 Mesos master 存在了一小部分持久化的状态，registry 的出现主要是为了解决下面两个问题：

- 如果 slave 在 master 故障时出错，那么任务丢失的通知会被发送给框架。这会导致框架和 master 之间的信息不一致——框架认为任务仍在运行，而 master 并不知道任务的存在。
- 如果一个新的 master 被选举出来，新的 slave 可以重新进行注册，这同样会造成框架和 master 之间的不一致——slave 上已经运行了一些任务，而框架并不了解。

目前 registry 中保存一个已经注册的 slave 列表，这样新选举出来的 master 可以检测出哪些 slave 没有重新注册，通知框架并阻止它们的重新注册。registrar 对 registry 上可执行的操作进行了抽象，隐藏了状态管理在底层不同状态存储的实现细节，并进行了诸如批量写的智能优化来避免可能的性能瓶颈。Mesos master 在注册、重新注册和 slave 删除的过程中通过 registrar 对 registry 进行操作。只有主 master 可以写入 registry，目前 Mesos 有如下几种 registrar 的后端实现。

- 内存：内存存储实现主要在 Mesos 的测试环境中使用，在正式环境中并不建议使用。
- LevelDB：LevelDB 是一个轻量级的持久化库（`http://leveldb.org`）。用户在非高可用模式下（即不需要和 ZooKeeper 集群通信的情况下），可以使用 LevelDB 作为存储后端。需要注意的是，如果需要将 Mesos master 移到另一台机器，那么同样需要移动在当前 master 工作路径下的 LevelDB 状态存储文件。
- Replicated log：Mesos 利用 MultiPaxos 算法来实现 replicated log。Replicated log 是一个只能进行追加操作的分布式数据结构，它可以存储任意数据。Replicated log 是 registry 默认的存储后端，也是推荐的存储后端。

扩展 Mesos

扩展性使得软件可以根据特殊需求进行调整，这对任何一个软件来说都十分重要。Mesos 一直试图将其核心进行最小化，将不是所有用户都需要的功能从核心中移除。Mesos 提供了集成接口，各种不同的外部系统实现都可以通过它与 Mesos 进行集成。几乎所有的 Mesos 组件都可以被不同的实现所替换，以完成特定的需求和功能。本书已经讨论过可插拔的隔离器、容器机及 registrar 实现，本节将会介绍更多允许用户定制化的 Mesos 功能。

Mesos 模块

Mesos 提供了模块化的架构，但是总会有一些特殊情况要求定制化 Mesos 来提供一些特定功能。Mesos 模块提供了无须重写、重新编译、重新链接即可扩展其内部功能的方式，使得 Mesos 可以应对定制化需求（`http://mesos.apache.org/documentation/latest/modules`）。举例来说，假设我们希望在 Mesos 上尝试一个用 Python 写的新分配算法。可以通过分配器模块来编写自己的模块，并让 master 加载新模块，而无须给 Mesos 增加新的依赖。通过这种方式，第三方软件可以很好地和 Mesos 集成，成为 Mesos 的扩展，这也提供了一种快速试验新特性的方法。

Mesos 目前支持多种模块。一类模块用来决定指定模块需要接收哪些回调，在 `src/examples` 包含了这些模块的样例。

- 鉴权模块：这类模块使得 Mesos 可以使用第三方的鉴权机制。Authenticatee 和 Authenticator 模块是这类模块的两个具体示例。
- 隔离器模块：这些模块可以提供特殊的隔离及监控机制。在示例中可以看到一个提供 GPU 资源隔离的模块。
- 匿名模块：这类模块不接受任何回调，它们只和其父进程共同存在。因此匿名模块并不会扩展或者替代 Mesos 中已经存在的功能。它们并不需要任何指定的配置项来发挥作用，而是在 master 或者 slave 初始化时被自动加载。
- 分配器模块：该模块在 Mesos master 中被定期调用来决定接下来每个框架需要获得怎样的资源。通过实现分配器模块，模块开发者可以提供新的分配机制，例如资源的超额认购及抢先占用。
- hook：并不是所有的 Mesos 内部组件都提供了完整合适的模块。模块开发者可以通过 hook 来扩展这些组件的功能。

模块作为共享库可以通过 `--modules` 选项指定模块名，并在 Mesos master 或者 slave 进程启动时进行加载。`--modules` 通过一个 JSON 字符串或者一个指向包含 JSON 字符串的文

件的文件路径来指定模块加载所需配置，典型的模块 JSON 配置模式如下所示：

```
{
  "type":"object",
  "required":false,
  "properties":{
    "libraries": {
      "type":"array",
      "required":false,
      "items":
      {
        "type":"object",
        "required":false,
        "properties":{
          "file": {
            "type":"string",
            "required":false
          },
          "name": {
            "type":"string",
            "required":false
          },
          "modules": {
            "type":"array",
            "required":false,
            "items":
            {
              "type":"object",
              "required":false,
              "properties":{
                "name": {
                  "type":"string",
                  "required":true
                },
                "parameters": {
                  "type":"array",
```

```
                "required":false,
                "items":
                {
                  "type":"object",
                  "required":false,
                  "properties":{
                    "key": {
                      "type":"string",
                      "required":true
                    },
                    "value": {
                      "type":"string",
                      "required":true
                    }
                  }
                }
              }
            }
          }
        }
      }
    }
  }
}
```

一个模块库中可以包含多个模块，模块的说明文件中需要为每个模块指定一个 `file` 或者 `path`。`file` 通过绝对或者相对路径指向机器上的一个位置。`name` 指定了模块名字，并根据所在平台对名字进行扩展名展开（在 Linux 平台扩展名为 `.so`，在 OS X 平台扩展名为 `.dylib`）。如果 `file` 和 `name` 两者都已被配置，那么 `file` 会被优先选择，`name` 会被忽略。如果没有指定路径，模块库会在平台的默认目录下（Linux 下为 `LD_LIBRARY_PATH`，OS X 下为 `DYLD_LIBRARY_PATH`）寻找相应的文件。下面是一个模块配置的简单例子：

```
{
  "libraries": [
    {
      "file": "/path/to/libab.so",
```

```
    "modules": [
      {
        "name": "org_apache_mesos_a",
        "parameters": [
          {
            "key": "X",
            "value": "Y",
          }
        ]
      },
      {
        "name": "org_apache_mesos_b"
      }
    ]
  }
]
}
```

上例中，从 /path/to/libab.so 文件中加载了两个指定的模块：org_apache_mesos_a 和 org_apache_mesos_b。其中 org_apache_mesos_a 在加载的过程中使用了 X=Y 选项，而 org_apache_mesos_b 模块在加载过程中没有使用任何选项参数。

Mesos 加载模块的过程分为如下几步：

1. 从模块实例中加载包含模块的动态库，这个实例可能是在命令行选项中所指定的。
2. 确认模块版本的兼容性。
3. 以单例模式初始化每个模块。
4. 将模块中的引用和实际所使用的函数进行绑定。

在本书写作时，Mesos 模块还是一个很新的功能（从 Mesos 0.21 版本才开始出现）并且仍然是一个试验特性。目前 Mesos 官方只维护 Mesos 核心模块（鉴权、隔离等）。关于 Mesos 模块的讨论目前在一个新建的单独邮件列表（modules@mesos.apache.org）中。

模块命名

Mesos 的每个模块都需要有一个唯一的名字。如果在模块的 JSON 配置文件中包含重复的名字，Mesos 进程将会异常退出。因此 Mesos 鼓励使用 Java 包命名规范（http://docs.oracle.com/javase/tutorial/java/package/namingpkgs.html）来对模块进行命名。通常情况下要遵守如下命名规范：

- 保持模块名的大小写规范一致。
- 使用翻转域名，并保持小写。
- 用下画线代替分隔符。
- 同一个组织下的不同模块需要有不同的名字。
- 不要用类型名称作为模块名。

例如，如果某个模块名为 fooBar，域名为 mesos.apache.org，那么模块的符号名称应该为 org_apache_mesos_fooBar。

模块兼容性

如之前所提到的，在加载模块前首先需要加载包含模块的动态链接库。为了保证兼容性，需要进行两步检查：

- 库兼容性检查，验证模块的 API 版本。
- 模块兼容性检查，检验模块类型版本。

模块的开发者需要指明所编写模块和哪个版本兼容。Mesos 在 src/manager/manager.cpp 中的一个表格中维护了兼容性信息。表格中包含了所有当前 Mesos 发行版本所能识别并兼容的模块类型。一个能成功通过兼容性检查的模块加载时需要保证这样的关系：类型版本号≤库版本号≤ Mesos 版本号。

分配模块

分配模块（http://mesos.apache.org/documentation/latest/allocation-module）负责决定哪个框架可以获得当前的资源 offer。框架可以拒绝不合适的资源 offer，并在合适的资源 offer 中运行任务。在框架通知任务结束、失败、由于 slave 故障导致任务丢失或者资源撤回的情况下，资源将被回收。分配模块是可插拔的，相关的 API 在 include/mesos/master/allocator.hpp 中进行了定义。

方法	描述
`initialize(flags, offerCallback,roles)`	初始化分配器
`addFramework(frameworkId, frameworkInfo, usedResources)` `removeFramework(frameworkId, frameworkInfo, usedResources)`	从分配机制中添加/删除框架
`activateFramework(frameworkId)` `deactivateFramework(frameworkId)`	激活/冻结指定框架，资源 offer 只能发送给处于激活状态的框架
`addSlave(slaveId, slaveInfo, totalResources, usedResources)` `removeSlave(slaveId, slaveInfo, totalResources, usedResources)`	从分配进程中添加/删除指定 slave
`activateSlave(slaveId)` `deactivateSlave(slaveId)`	激活/冻结指定的 slave。只有从激活状态的 slave 发送来的资源 offer 才会被分配器进行分配
`requestResources(frameworkId, requests)`	当框架接收指定的资源请求时所触发的回调函数
`updateAllocation(frameworkId, slaveId, operations)`	更新指定框架在指定 slave 上所分配的资源
`recoverResources(frameworkId, slaveId, resources, filters)`	从指定框架回收资源时所触发的回调函数
`reviveOffers(frameworkId)`	框架想要重新获得之前所排除的资源 offer 所触发的回调函数
`updateWhitelist(whitelist)`	更新 slave 的白名单

分配器模块由 C++ 实现，并且必须实现在 `mesos/master/allocator.hpp` 文件中定义的所有接口。如果想用其他编程语言来实现自定义分配器，必须利用 C++ 实现一个简单的代理，用来委托处理另一个语言实现的分配器的调用请求。自定义的分配器可以在 Mesos master 上通过 `--modules` 参数和 `--allocator` 参数来指定加载。

在 `src/master/allocator/mesos/hierarchical.hpp` 中实现的 `HierarchicalDRFAl`

locatorProcess 是 Mesos 的默认分配器。基于 actor 的实现使得它是非阻塞的，方法调用在将消息放入指定 actor 的消息队列后会立即返回。可以复用在 src/master/allocator/mesos/allocator.hpp 中 MesosAllocatorProcess 所实现的 actor 设计，并且将自己设计的基于 actor 的分配器打包进 MesosAllocator 中。

另一种修改资源分配的方法是为内建的层级分配器实现新的排序算法。分配模块通过排序算法来确定给客户（用户角色和框架）分配资源的顺序。排序器可以遵循 src/master/allocator/sorter/sorter.hpp 中所定义的排序 API，用 C++ 来实现。

方法	描述
`void add(client, weight=1)` `void remove(client)`	从分配算法中增加/删除某个客户
`void deactivate(client)` `void activate(client)`	从排序器中删除/增加某个客户

`void add(slaveId, resources)`
`void remove(slaveId, resources)`
`void update(slaveId, resources)`
	增加/删除/更新资源池中所拥有的总资源

`List<string> sort()`	返回依照排序算法重新排序后的客户列表，分配器需要按照这个列表的顺序进行资源分配

`void allocated(client, slaveId, resources)`
`void update(client, slaveId, oldResources, newResources)`
`void unallocated(client, slaveId, resources)`
	将资源从指定客户处分配/升级/回收

`Map<SlaveId, Resources> allocation(client)`	返回分配给某个客户的资源
`bool contains(client)`	如果排序器包含指定的客户，则返回 true
`int count()`	返回排序器所包含的客户数量

分层分配器使用的默认排序器是 DRFSorter，在 src/master/allocator/sorter/drf/sorter.hpp 中可以找到对应的实现。它实现了资源的公平共享并支持带权优先级。

Mesos hook 和修饰器

hook 提供了另一种扩展 Mesos 内部工作的方式，但是和模块不同，hook 并不会修改处理请求的过程。hook 是 Mesos 各个组件在执行生命周期的不同阶段会去调用的函数。它们提供了一种在 Mesos 生命周期中注入功能的方法。这也使得外部工具和系统可以更容易地同 Mesos 进行集成。可以将 hook 当成分布在 Mesos 各个组件之中的事件回调函数。在本书写作时，Mesos 0.23 版本已经支持任务启动的 hook。Mesos 的 hook API 在 `mesos/hooks.hpp` 中定义，并定义了在各种上下文环境中不同 hook 的切入点。例如，`masterLaunchTaskHook` 会将 `TaskInfo` 作为上下文环境的一部分。修饰器是一种拥有返回类型的特殊 hook，它可以在对象不同生命周期切换的过程中对其进行修改。例如，修饰器可以在 `TaskInfo` 传入 Mesos master 的时候添加新的任务标签。Mesos 运行所需要的 hook 通过一个由逗号分隔的列表在 `--hooks` 选项中指定。

任务标签

标签在和其他一系列服务系统（如服务发现、外部安全系统、资源监控和审计系统）进行集成时十分有用。在 0.22 版本之前，Mesos 中没有标签机制，只能通过不同的任务名称的模式来区分这些不同主机。任务标签使得可以为每个任务提供一个全局可见的元信息。Mesos 本身不会对标签信息进行处理，而是由外部工具来实现对标签处理的工作。标签信息通过 master 和 slave 接入点提供，外部工具可以通过这些接入点来请求标签信息。

标签可以是任意附加在 Mesos `TaskInfo` 中的键值对。需要注意区分的是 `TaskInfo` 中有一个 `data` 域，它也可以存储任意类型的数据，但是这个域通常用来保存和执行器相关的一些信息。此外和 `data` 域不同，标签信息通常保存在 master 和 slave 的进程内存中。因此标签中只能记录一些关于任务的轻量级信息。

小结

本章深入探讨了 Mesos 架构中的细节，并了解了 Mesos 如何处理资源分配、资源隔离及容错。还介绍了对 Mesos 进行扩展的多种方式。

下一章将介绍如何开发一个 Mesos 框架。

第7章 开发 Mesos 框架

Mesos 是一个通用的计算集群的基础。这意味着 Mesos 可以支持大量使用通用资源的框架。上一章已经介绍了如何运行多种不同的框架。在 Mesos 社区中还有更多框架，它们满足了更多的特殊化需求和不同的工作负载，同时也验证了 Mesos API 的通用性。在现实中总会遇到那些现有框架无法处理的需求，因此是时候来介绍如何创造一个框架，从而充分利用 Mesos 的能力满足我们自己的需求了。本章将会讨论 Mesos API 的细节信息，并将一步步利用 Java 实现一个自己的 Mesos 框架。本章将会介绍如下主题：

- Mesos API
- 开发一个 Mesos 框架
- 创建自己的框架
- 高级主题
- 开发者资源

Mesos API

Mesos 提供了简单明了的 API 并隐藏了底层和分布式交互的具体细节，使得我们可以在分布式系统上来开发自己的应用。Mesos API 的细节讨论可以参考 `http://mesos.apache.org/documentation/latest/app-framework-development-guide/`。在进一步深入 Mesos API 前，首先来看 Mesos 所使用到的各种消息。

Mesos 消息

Mesos 使用 protocol buffer 来定义发送给不同组件的消息。Protocol buffer 定义了可以跨平台和编程语言的消息格式。下面介绍的不同 Mesos 组件之间交互的消息格式可以在 `include/mesos/mesos.proto` 找到对应的定义。

- `MasterInfo`：用来描述 master 信息，包括 ID、主机名、IP、端口及 master PID。
- `SlaveInfo`：用来描述 slave 信息，包括 ID、主机名、IP、端口及资源和属性信息。
- `Resource`：描述了一台机器上的资源信息。
- `Attribute`：描述了一台机器上的属性信息。
- `Request`：描述了资源请求信息。它由描述一台机器上资源信息的 resource 域，以及指明资源请求来自哪个 slave 的 `slaveId` 域组成。
- `FrameworkInfo`：描述框架信息。它由主机名、用户名、指明角色和权限的鉴权信息组成。此外还可以通过 FrameworkInfo 访问到其他的框架配置参数，如 `failover_timeout`、checkpoint 及框架的 `webui_url`。如果 `frameworkId` 域被设置了，那么调度器会根据该值进行故障恢复。
- `CommandInfo`：描述了命令和执行 shell 的信息。它包含了指定文件所在位置的下载 URI。`extract` 域指明在文件下载结束后是否要进行解压缩操作。如果该域被设置，文件会被解压到执行器所在的工作路径。目前该域的默认值为 true，`CommandInfo` 支持 `tar.gz` 和 `zip` 两种压缩格式。如果 executable 域被设置，那么下载的文件会被赋予可执行权限。`CommandInfo` 中还在 `ContainerInfo` 和 `Environment` 域中包含了运行时所需要的容器信息和环境信息。`ContainerInfo` 中包含了镜像 URI 以及传递给容器机的参数。`user` 域指定了需要以哪个用户身份来运行命令，`auguments` 域包含了命令运行的参数。对于 `shell` 域，如果设置为 true，那么 command 所包含的值将会以 `/bin/sh -c` 的方式来执行，并且忽略所有 argument 域的参数；如果该域设置为 false，那么所有参数会传递给 command 来执行。
- `ContainerInfo`：用来描述外部容器配置。它包括主机名、卷信息及类型。`Volume` 描述了从主机到容器的路径映射、主机上的路径（绝对路径或者相对于容器工作路径的相对路径）和挂载的模式（可读写或者只读）。`Type` 域描述了容器的类型，当前可选的类型为 `Mesos` 和 `Docker`。可选的 `DockerInfo` 域描述了 Docker 容器运行所需的配置信息，包含了镜像信息、网络配置信息（host、bridge 或者 none）、主机和容器的端口映射信息，以及其他需要传递给 docker CLI 的参数；还包含了一个标志位，用来指明是否要在特权模式下运行 Docker 并且强制从仓库中下载 Docker 镜像。

- DiscoveryInfo：包含了服务发现所需要的信息。Visibility 域用来限制执行器服务发现的范围。可能的值有 FRAMEWORK、CLUSTER 和 EXTERNAL，可见范围为 FRAMEWORK 的任务只能被运行在相同框架下的其他任务发现；可见范围为 CLUSTER 的任务可以被所有 Mesos 集群中运行的任务所发现；可见范围为 EXTERNAL 的任务可以被 Mesos 之外的第三方服务和客户所发现。
- ExecutorInfo：用来描述执行器的信息。它由 executorID、frameworkID、CommandInfo、containerInfo、discoveryInfo、resources，以及配置信息如 name、source 等组成。Data 域可以用来传递任意数据。
- HealthCheck：用来描述对任意命令、进程、任务及执行器的健康检查信息，以及其他的配置参数，如超时参数、间隔时间、静默时间等。
- ResourceUsage：描述了某个执行器在某一时刻的资源使用情况，它由 slaveId、frameworkId、executorId、executorName、taskId 和包含资源使用详细信息的 ResourceStatstics 组成。
- Offer：描述 slave 上的可用资源，由 offerId、hostname、slaveId、framework Id、executorId、attribute 和 resource 域组成。
- TaskInfo：用来描述任务的信息，由 name、taskId、SlaveId、resource、executorInfo、CommandInfo、containerInfo、discoveryInfo、HealthCheck、data 和 Labels 域组成。Labels 域是任意的键值对，可以包含任何和框架相关的信息。
- TaskStatus：描述任务当前状态信息，由 taskId、taskState、slaveId、executorId、可选的 message 和 data 域组成。TaskState 是一个枚举类型，用来描述任务可能的状态，如等待、启动、运行、结束、失败、丢失等。
- Filters：用来描述框架调度器对不可用资源进行过滤的信息，使用 refuse_seconds 来指定资源未被使用的时长界限，超过这一界限就拒绝当前的资源。该项默认值为 5 秒。

调度器 API

框架调度器负责管理框架所获得的资源。框架从 Mesos master 获取资源并负责响应和关联执行器的资源请求。org.apache.mesos.Scheduler 接口定义了框架从 Mesos master 接收回调所必须实现的接口。由于所有的方法都可以访问驱动实例，执行器本身不需要进行存储。调度器所要实现的接口可以在 MESOS_HOME/include/mesos/scheduler.hpp 中找到，下面对其中定义的方法进行介绍。

- `registered(SchedulerDriver, FrameworkID, MasterInfo)`：该方法在调度器向 Mesos master 注册时被调用。`FrameworkId` 是由 Mesos 所提供的全局唯一 ID。这个方法可以用来初始化任何调度器数据结构。
- `reregistered(SchedulerDriver, MasterInfo)`：该方法在框架调度器向新选举的 master 重新注册后调用，`masterInfo` 中提供了新 master 相关的信息。
- `disconnected(SchedulerDriver)`：该方法在调度器和 master 断开连接时被调用。
- `resourceOffers(SchedulerDriver, List<Offer>)`：这是所有调度器接口中最重要的一个。`resourceOffers()` 在 master 向框架提供资源 offer 时调用。每一份资源 offer 包含从某个 slave 上获取的资源列表。调度器可以接受资源 offer 并利用 `offerIds` 来启动任务，也可以拒绝资源 offer。根据不同的分配器实现，一个资源 offer 可以提供给多个框架；在这种场景下，第一个使用资源 offer 启动任务的框架将会胜出，其他框架将会接收到 `offerRescinded()` 发出的消息。
- `offerRescinded(SchedulerDriver, OfferID)`：该方法在 master 撤销发送给框架的资源 offer 时被调用。这可能在 slave 消失或者其他框架已经使用了该资源 offer 的情况下发生。如果调度器没有接收到回调并且尝试利用已经过期的 `offerId` 启动任务，那么将会从任务接收到 `TASK_LOST` 的状态。
- `statusUpdate(SchedulerDriver, TaskStatus)`：每当 Mesos 向框架发送消息时该方法都会被调用。在任务状态发生变化时，如任务结束或者由于 slave 丢失而导致任务丢失时，该方法会被调用。可以使用该方法来跟踪框架下工作的进度，例如标记任务结束，在失败时重新启动或者忽略状态更新。
- `frameworkMessage(SchedulerDriver, ExecutorID, SlaveID, buyte[] data)`：该方法用来向调度器传递执行器发送的消息。调度器可以访问执行器和 slave 的 ID，以及调度器所发送的数据。消息传输采用基于最大努力交付服务，消息在遇到失败事件时并不会重新传输。
- `slaveLost(SchedulerDriver, SlaveID)`：该方法用来通知调度器 Mesos 无法和指定 ID 的 slave 进行通信。收到该消息的典型响应是将之前在 slave 上运行的任务调度到另一个新的 slave 上重新执行。
- `executorLost(SchedulerDriver, ExecutorID, SlaveID,int status)`：该方法用来通知调度器特定的执行器已经以指定状态结束了。对于大多数框架而言，调度器无须额外的特殊操作，因为 `TASK_LOST` 状态在执行器运行任务时会被更新。
- `error(SchedulerDriver, String message)`：该方法在一个可恢复的错误产生或者调度器和驱动停止工作时被调用。该方法应该清理所有使用到的资源，如锁和其他资源。

调度器驱动 API

调度器驱动接口定义了调度器生命周期函数及其 Mesos 交互的方法。它负责调用框架调度器的回调函数。

所有方法都在调用结束后返回一个 `Status` 结构。下面是调度器生命周期不同阶段的方法。

- `start()`：该方法启动调度器，它必须在其他方法被调用前执行。
- `stop(boolean failover)`：该方法用来停止一个驱动。failover 标志位用来表明框架是否设置了故障恢复。如果该标志位为 false，那么框架不会和 Mesos 重新连接，Mesos 会注销该框架并杀死该框架下所有运行的执行器和任务。如果该标志位为 true，那么 Mesos 会继续运行执行器和任务，并允许驱动重新和它们建立连接。新的驱动可能在相同的进程中运行，也可能在不同机器的不同进程上运行。stop 方法还有一个无参数的版本 `stop()`，它和 `stop(false)` 是等价的。
- `abort()`：该方法会终止 driver，并停止所有发送给调度器的回调。在特定情况下（开启故障恢复）将会启动一个新的 driver 实例。
- `join()`：该方法等待驱动退出（`stop()` 或者 `abort()`）并有可能一直阻塞下去。通过它的返回状态可以知道驱动是正常退出的还是异常退出的。
- `run()`：该方法是依次执行 `start()` 和 `join()` 的一个便利方法。
- `requestResource(Collection<Request>)`：该方法向 Mesos 请求资源，并将资源提供给资源框架调度器。
- `launchTask(Collection<OfferID> offerIds, Collection<TaskInfo> tasks)` 和 `launchTask(Collection<OfferID> offerIds, Collection<TaskInfo> tasks, Filters filters)`：该方法在一组资源 offer 上启动一组任务。没有被执行器和任务所使用的资源 offer 视为被拒绝。如果有过滤器的存在，过滤器将在剩余资源上执行过滤。如果没有指定任务需要执行，所有的资源都视为被拒绝。
- `declineOffer(OfferID offerId, Filters filters)` 和 `declineOffer(offerID offerId)`：该方法用来拒绝某个 offer 的资源，如果 `Filters` 存在，则它也会被使用。
- `reviveOffers()`：该方法删除之前设置的过滤器，使得框架可以从之前被过滤的 slave 中接受资源 offer。
- `sendFrameworkMessage(ExecutorID executorId, SlaveID slaveId, byte[] data)`：该方法用来从框架向执行器发送消息。消息传输采用基于最大努

力交付服务，消息传输失败时并不会重新传输。

- `reconcileTasks(Collection<TaskStatus> statuses)`：调用该方法时，master会针对参数中所指定的任务将其上次接受到的任务状态送回。如果参数没有指定任务，那么master会发送其知道的所有非终止态的任务状态。框架可以利用这个方法来搜索任务状态。那些没有被发现的任务会被更新为`TASK_LOST`状态。

`MesosDriverScheduler`是`DriverScheduler`接口的一个实现，其通过Java本地方法来调用Mesos的原生实现。我们将使用它来和Mesos进行交互。所有的`MesosDriverSheduler`方法都是线程安全的，由于其和调度器运行在不同的线程中，驱动器中方法的阻塞不会对调度器的回调产生任何影响。

执行器API

框架执行器负责启动任务并执行调度器所分配的任务。`org.apache.mesos.Executor`定义了框架执行器所必须实现的接口，可以在`MESOS_HOME/include/mesos/executor.hpp`找到相关的细节信息。由于每次只能进行一次回调，所以执行器不能被阻塞。执行器能够访问其所在的`ExecutorDriver`实例，并且可以使用`MESOS_HOME`变量来寻找Mesos所在的位置。

- `registered(ExecutorDriver, ExecutorInfo, FrameworkInfo, SlaveInfo)`：该方法在执行器驱动成功和Mesos连接后调用。调度器可以利用`ExecutorInfo`中的data域将任意数据传递给执行器。`FrameworkInfo`包含了框架相关的信息，`SlaveInfo`中包含了将要运行执行器的slave。
- `reregistered(ExecutorDriver, SlaveInfo)`：该方法在执行器向重启的slave重新注册时调用。
- `disconnected(ExecutorDriver)`：该方法在执行器失去和slave的连接，例如slave重启时调用。
- `launchTask(ExecutorDriver, TaskInfo)`：该方法是执行器最核心的方法。当任务在当前执行器上启动时，`launchTask()`会被调用。需要注意，和执行器的其他方法一样，该方法在调用时也会阻塞，在回调完成前，该执行器将无法执行其他回调函数。因此如果需要运行长时间的任务，最好将其放在一个单独的线程中运行。任务可以通过线程或者进程的方式启动。
- `killTask(ExecutorDriver, TaskID)`：该方法在当前执行器中任务被杀死时调用。需要注意的是，执行器并不会进行任何的状态更新，如果执行器想要捕获状态的变化，就必须自己产生并发送一个针对该任务的状态更新，将其置为

TASK_KILLED。

- frameworkMessage(ExecutorDriver, byte[] data)：该方法在框架消息到达执行器时被调用。消息传输采用基于最大努力交付服务，消息在遇到失败事件时并不会重新传输。
- shutdown(ExecutorDriver)：该方法用来通知执行器结束所有运行中的任务。如果执行器没能成功发送任何任务结束的更新（TASK_KILLED、TASK_FINISHED 或者 TASK_FAILED），那么一个 TASK_LOST 的状态更新将会被创建。
- error(ExecutorDriver driver, String message)：该方法在执行器或执行器驱动中发生致命性错误时被调用。这里的 message 参数中包含了错误的细节信息。注意驱动会在调用该回调函数前异常退出。

调度器驱动 API

ExecutorDriver 是连接执行器和 Mesos 的接口。接口中包含了执行器在生命周期各阶段的方法，以及向 Mesos 发送消息的方法。下面是接口中所包含的所有方法，和 SchedulerDriver 方法类似，所有的 ExecutorDriver 方法都会返回当前驱动的状态。

- start()、stop()：start() 和 stop() 方法用来对驱动进行初始化及清理。start() 函数需要在所有其他函数被执行前调用。
- abort()：该方法在驱动异常退出，任何回调都无法发送给执行器时被调用。和 stop() 不同，abort() 后新的驱动实例可以在旧实例所在的进程中启动。
- join()：该方法用来等待驱动停止或者异常退出返回给 join 的状态码，状态码可以用来区分驱动调用的究竟是 stop() 还是 abort()。
- run()：该方法在启动驱动并且阻塞后立刻调用 join()。
- sendStatusUpdate(TaskStatus status)：该方法用来向调度器发送状态更新消息。该方法在收到消息确认前会不断重试。如果执行器异常退出，TASK_LOST 的状态更新将会发送给调度器。
- sendFrameworkMessage(byte[] data)：该方法用来向调度器发送消息。消息传输采用基于最大努力交付服务，消息传输失败时并不会重新传输。

MesosExecutorDriver 提供了内建的 ExecutorDriver 接口实现。可以利用它来调用执行器的回调函数，以便和 Mesos slave 进程进行通信。值得一提的是，MesosExecutorDriver 是线程安全且非阻塞的，它不会影响其他的回调函数执行。

值得注意的是，Mesos Scheduler API 和 Executor API 都是 Mesos 核心的一部分，而 SchedulerDriver API 和 ExecutorDriver API 都是通过语言绑定来提供的。

开发一个 Mesos 框架

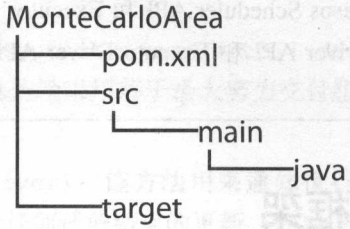
一个 Mesos 框架由调度器、执行器和启动器组成，其中执行器是非必需的，可以根据情况省略之。我们将依照下面的步骤来开发一个自己的框架：

1. 搭建开发环境。
2. 加入框架调度器。
3. 加入框架启动器。
4. 编写执行器。
5. 编译打包。
6. 在集群中安装框架。

搭建开发环境

Mesos 框架开发环境的搭建并不需要特殊步骤。通过任何 IDE 编辑器甚至是终端就可以完成。所需要做的只是访问语言绑定来和 Mesos 进行交互。Mesos 提供了几乎所有语言的绑定。当前的 Mesos 发行版提供 Java 和 Python 的绑定，社区正在开发针对 Go、Erlang、Haskell 和 Clojure 的绑定。完整的语言绑定和其他 Mesos 相关资料可以在 `https://github.com/dharmeshkakadia/awesome-mesos/` 中找到。这些语言绑定可以分为两类：纯净版和非纯净版。纯净版绑定通过 HTTP 协议和 Mesos 进行通信。非纯净版绑定需要在机器上安装 `libmesos` 来和 Mesos 进行通信。

下文将使用 Mesos 的 Java API 来编写框架。利用其他语言的绑定来编写框架和使用 Java 客户端绑定十分类似。如果使用非纯净版的绑定，那么需要首先在开发机器上安装 Mesos，或者和远程的 Mesos 集群通信。正如之前所介绍的，在编译和运行时需要一些客户端的类库。可以通过依赖管理工具的帮助来处理这些事情。本文使用 Maven 来处理构建和打包的相关事宜。完整的上手模板代码可以在 `https://github.com/dharmeshkakadia/MonteCarloArea/` 找到。本文将创建一个用来估算曲线与矩形框所包围面积的框架，并将这个框架命名为 `MonteCarloArea`。框架的细节将在下一章详细介绍。代码的目录结构如下所示：



src/main/java 中包含了源代码文件，target 目录中包含编译后的二进制文件，根目录的 pom.xml 文件用来描述 Maven 依赖。

现在给 Mesos 框架加入少量代码骨架。我们从给项目加入 Mesos 客户端类库开始。Mesos Java 绑定可以通过 Maven 的中心仓库获得。将下面的配置加入 pom.xml 文件就可以将最新版本的绑定加入到项目中。

```
<dependencies>
    <dependency>
        <groupId>org.apache.mesos</groupId>
        <artifactId>mesos</artifactId>
        <version>0.20.1</version>
    </dependency>
</dependencies>
```

加入框架调度器

我们的第一个调度器类是 org.packt.mesos.MonteCarloScheduler。我们将要在其中实现调度器接口，首先只是简单地打印消息而不增加任何回调逻辑，接下来再不断加入和计算相关的调度器及执行器逻辑。

```
public class MonteCarloScheduler implements Scheduler {

    public MonteCarloScheduler(){
    }

    @Override
    public void registered(SchedulerDriver schedulerDriver,
            Protos.FrameworkID frameworkID, Protos.MasterInfo masterInfo) {
        System.out.println("Scheduler registered with id " +
```

```
        frameworkID.getValue());
    }

    @Override
    public void reregistered(SchedulerDriver schedulerDriver,
            Protos.MasterInfo masterInfo) {
        System.out.println("Scheduler re-registered");
    }

    @Override
    public void resourceOffers(SchedulerDriver schedulerDriver,
            List<Protos.Offer> offers) {
        System.out.println("Scheduler received offers " +offers.size());
    }

    @Override
    public void offerRescinded(SchedulerDriver schedulerDriver,
            Protos.OfferID offerID) {

    }

    @Override
    public void statusUpdate(SchedulerDriver schedulerDriver,
            Protos.TaskStatus taskStatus) {
        System.out.println("Status update: task " +
            taskStatus.getTaskId().getValue() + " state is " +
            taskStatus.getState());
    }

    @Override
    public void frameworkMessage(SchedulerDriver schedulerDriver,
            Protos.ExecutorID executorID, Protos.SlaveID slaveID,
            byte[] bytes) {

    }
```

```
    @Override
    public void disconnected(SchedulerDriver schedulerDriver) {

    }

    @Override
    public void slaveLost(SchedulerDriver schedulerDriver,
            Protos.SlaveID slaveID) {

    }

    @Override
    public void executorLost(SchedulerDriver schedulerDriver,
            Protos.ExecutorID executorID, Protos.SlaveID slaveID, int i) {

    }

    @Override
    public void error(SchedulerDriver schedulerDriver, String message) {
    }
}
```

MonteCarloScheduler.resourceOffers() 是调度器响应 Mesos master 发送的资源 offer 的主要方法。目前调度器没有做任何事情。部署这个空的调度器使得可以通过它看到各个不同事件发生时产生的消息。

加入框架启动器

框架启动器负责创建框架驱动实例，并对框架整个生命周期进行管理。我们将会调用自己编写的名为 App 的启动器类，它只有一个 main 方法：

```
public class App {
    public static void main(String[] args) {
        System.out.println("Starting the MonteCarloArea on Mesos with master "
            + args[0]);
        Protos.FrameworkInfo frameworkInfo =
```

```
        Protos.FrameworkInfo.newBuilder().setName("MonteCarloArea")
            .build();
        MesosSchedulerDriver schedulerDriver =
            new MesosSchedulerDriver(new MonteCarloScheduler(),
                frameworkInfo, args[0]);
        schedulerDriver.run();
    }
}
```

这里使用 MesosScheculerDriver 来进行生命周期管理，MesosSchedulerDriver 的构造函数需要一个框架调度器实例、一个描述框架信息的 FrameworkInfo 实例，以及 Mesos master 的地址来进行初始化。Mesos master 的地址可以以任何 Mesos 可接受的形式来指定，例如 host:port、zk://host1:port1/path/to/mesos/ 或者 file://path/to/file/containing/master/URI；也可以传入可选的框架证书信息。FrameworkInfo 只是用来设置框架的名称。最后利用 run() 函数来启动并阻塞驱动。

部署框架

部署框架需要如下三个步骤：

1. 编译打包框架。
2. 安装框架。
3. 启动框架。

下面我们来构建代码，首先可以从 IDE 或者命令行发起一个 mvn 的 build 命令，接下来运行下面的命令：

```
ubuntu@master:~ $ mvn package
```

这将会编译整个框架的代码并在 target/ 目录下生成 jar 文件。当构建模块成功，将模块在 Mesos 集群进行安装后，需要让所有的 slave 都可以访问到新生成的框架二进制文件。可以将二进制文件放在 HDFS 上，利用 MesosSchedulerDriver 构造函数中的 ExecutorInfo 参数来告知 Mesos 在什么地方去寻找这个文件。也可以将执行器的 URI 域设置为指向 HDFS 上路径的 URI；还可以在 Mesos slave 上指定 frameworks_home 参数，这个路径下存储了所有的框架和执行器。本例中，ExecutorInfo 可以被设置为相对路径，frameworks_home 的值将成为相对路径的前缀。另一个方法是利用 Docker 来打包执行器以及包含相应的运行参数的 DockerInfo。

为了运行框架，需要在 classpath 中加入 Mesos jar 及 protocol buffer jar。一个选择是构建一个包含了所有依赖的体积庞大的 jar 包，另一个选择是在运行时将它们动态地加入 classpath 中，本例使用第二种方法。可以利用 Chef 或者 shell 脚本作为部署工具来启动框架。此外还可以利用之前章节提到的元框架来启动框架。当我们想在一个节点启动一条命令时，类似于 mesos-submit（`http://mesos.apache.org/documentation/using-the-mesos-submit-tool/`）的工具十分有用。如果没有这样的工具，必须在开发者的机器上保持一个调度器程序的运行。mesos-submit 也是一个框架，它可以创建执行器并启动真实的调度器来接管当前框架调度器的工作。至此，mesos-submit 可以安全退出了。

利用下面的命令从命令行启动框架：

```
ubuntu@master:~ $ java -cp MonteCarloArea.jar:mesos-0.20.1.jar:protobuf-2
.5.0.jar -Djava.library.path=libs/ org.packt.mesos.App "zk://master:2181/
mesos"
```

现在我们拥有了特定 protocol buffer 的 jar 文件及 Mesos Java 绑定。还需注意的是我们的 Java 绑定用到了 Mesos 的内建实现，因此需要知道 `libmesos.so` 所在的位置。在上面的命令中，它在 `libs` 路径下。此外还需要指定启动器的全名 `org.packt.mesos.App`。最后一个需要的参数是 Mesos master 的地址。

现在我们的框架调度器已经开始运行了，你将会看到类似下面这样的输出：

```
I1005 14:06:28.488776  7710 sched.cpp:391] Framework registered with
20141005-130347-1739597066-5050-1453-0005
Scheduler registered with id 20141005-130347-1739597066-5050-1453-0005
...
Scheduler received offers 2.
...
I1005 14:06:29.381525  7704 sched.cpp:730] Stopping framework
'20141005-130347-1739597066-5050-1453-0005'
```

框架的注册和其他事件可以在 master 的输出中看到：

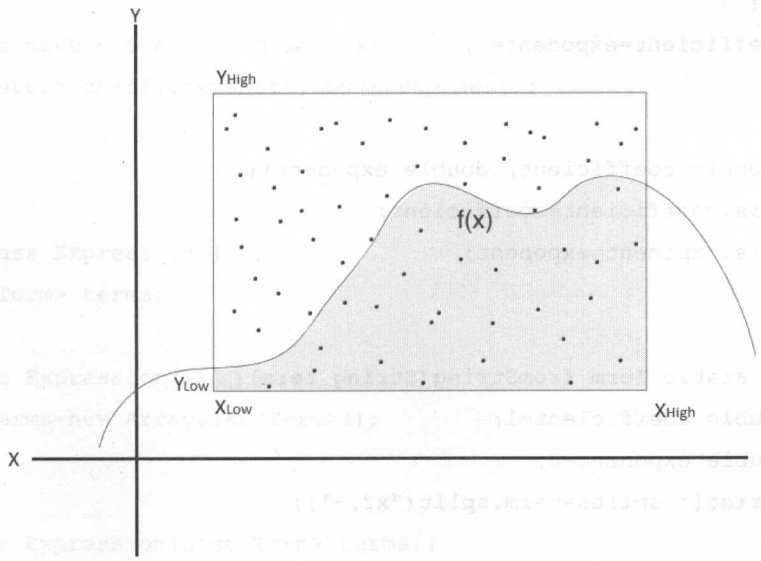
```
I1005 14:06:28.488001  1461 master.cpp:818] Received registration request
from scheduler(1)@10.37.176.103:54774
I1005 14:06:28.488085  1461 master.cpp:836] Registering framework
20141005-130347-1739597066-5050-1453-0005 at scheduler(1)@10.37.176.103:
54774 I1005 14:06:28.488234  1461 hierarchical_allocator_process.hpp:332]
Added framework 20141005-130347-1739597066-5050-1453-0005
```

```
I1005 14:06:28.488368  1461 master.cpp:2285] Sending 1 offers to framework
20141005-130347-1739597066-5050-1453-0005
...
I1005 14:06:29.382316  1459 master.cpp:1034] Asked to unregister framework
20141005-130347-1739597066-5050-1453-0005
I1005 14:06:29.382344  1459 master.cpp:2688] Removing framework
20141005-130347-1739597066-5050-1453-0005
I1005 14:06:29.382439  1459 hierarchical_allocator_process.hpp:408]
Deactivated framework 20141005-130347-1739597066-5050-1453-0005
I1005 14:06:29.382472  1459 hierarchical_allocator_process.hpp:363]
Removed framework 20141005-130347-1739597066-5050-1453-0005
```

网页控制台中也会展示框架，slave 的日志信息也包含框架所发送消息状态的列表。可以通过按 Ctrl + C 键退出框架，关闭框架相关的消息也会被展示出来。

构建框架

我们的框架工作十分简单，计算直角坐标系和曲线所包围的曲线面积。我们通过数值计算中的蒙特卡洛算法来计算曲线积分，下图展示了面积计算的思路。



平行坐标轴的矩形框及函数 $f(x)$

首先在矩形框内分布大量的点，接下来计算曲线函数在每个点 x 坐标处的值。如果曲线函数的值大于该顶点 y 的坐标，那么该点处于曲线下方。统计所有这样点的数量，那么曲线所包围的面积就可以通过下述表达式进行计算：

$$\text{曲线包围面积} = \text{矩形框面积} \times (\text{曲线下顶点数量} \div \text{总顶点数量})$$

灰色区域的面积大小就是所需的结果。

这种方式计算结果的准确程度依赖于分布点的数量，分布点的数量越多，计算结果越准确。但是，随着分布点数量的增加，计算的过程对于单台机器来说成为资源紧张的任务，因此需要更多的计算资源。我们可以将大问题切分成小问题，在这里可以把整个矩形框切分成多个小矩形框，然后利用 Mesos 的 API 去使用多台机器的计算资源。

这里利用线性方程来表示曲线，方程的每一项通过一个字符串来表示。Term 类用来描述方程的每一项，表达式用来描述曲线，我们利用四个坐标 X_{Low}、X_{High}、Y_{Low}、Y_{High} 来描述平行于坐标轴的矩形框：

```
class Term{
    double coefficient;
    double exponent;

    Term(){
        coefficient=exponent=0;
    }

    Term(double coefficient, double exponent){
        this.coefficient=coefficient;
        this.exponent=exponent;
    }

    public static Term fromString(String term){
        double coefficient=1;
        double exponent=0;
        String[] splits=term.split("x",-1);

        if(splits.length>0) {
            String coefficientString=splits[0].trim();
```

```
            if(!coefficientString.isEmpty()) {
                coefficient = Double.parseDouble(coefficientString);
            }
        }

        if (splits.length>1) {
            exponent=1;
            String exponentString = splits[1].trim();
            if (!exponentString.isEmpty()) {
                exponent = Double.parseDouble(exponentString);
            }
        }
        return new Term(coefficient, exponent);
    }

    @Override
    public String toString() {
        return coefficient+"x^"+exponent;
    }

    public double evaluate(double x){
        return coefficientMath.pow(x,exponent);
    }
}

public class Expression {
    List<Term> terms;

    public Expression(){
        terms=new ArrayList<Term>();
    }

    public Expression(List<Term> terms){
        this.terms=terms;
    }
```

```
    public boolean addTerm(Term term){
        return terms.add(term);
    }

    public double evaluate(double x){
        double value=0;
        for (Term term : terms) {
            value+=term.evaluate(x);
        }
        return value;
    }

    public static Expression fromString(String s){
        Expression expression=new Expression();
        String[] terms = s.split("\+");
        for (String term : terms) {
            expression.addTerm(Term.fromString(term));
        }
        return expression;
    }

    @Override
    public String toString() {
        StringBuilder builder=new StringBuilder();
        int i;
        for (i=0; i<terms.size()-1; i++) {
            builder.append(terms.get(i)).append(" + ");
        }
        builder.append(terms.get(i));
        return builder.toString();
    }
}
```

Term 中的 `fromString()` 函数用来解析给定的方程项。需要注意的是我们的目标并不是写一个复杂的解析器，而是写出一个可能在一些边缘状况下无法正常工作的简单版本。Term

中的 evaluate() 方法用来计算在给定 X 值情况下，方程某一项的值是多少。expression 中的 evaluate() 方法用来计算在指定 X 值下，所有项的值的总和。

给框架加入执行器

下面让我们加入计算矩形框 AUC 的执行器，执行器的名称为 org.packt.mesos.MonteCarloExecutor，其实现了所有的执行器接口，并保存了表达式、矩形框坐标，以及分布点的个数：

```
public class MonteCarloExecutor implements Executor{
    Expression expression;
    double xLow;
    double xHigh;
    double yLow;
    double yHigh;
    int n;

    public MonteCarloExecutor(Expression expression, double xLow,
            double xHigh, double yLow, double yHigh, int n) {
        this.expression = expression;
        this.xLow = xLow;
        this.xHigh = xHigh;
        this.yLow = yLow;
        this.yHigh = yHigh;
        this.n=n;
    }

    @Override
    public void registered(ExecutorDriver executorDriver,
            Protos.ExecutorInfo executorInfo,
            Protos.FrameworkInfo frameworkInfo,
            Protos.SlaveInfo slaveInfo) {
        System.out.println("Registered an executor on slave " +
            slaveInfo.getHostname());
    }
```

```
    @Override
    public void reregistered(ExecutorDriver executorDriver,
            Protos.SlaveInfo slaveInfo) {
        System.out.println("Re-Registered an executor on slave " +
            slaveInfo.getHostname());
    }

    @Override
    public void disconnected(ExecutorDriver executorDriver) {
        System.out.println("Re-Disconnected the executor on slave");
    }

    @Override
    public void launchTask(final ExecutorDriver executorDriver,
            final Protos.TaskInfo taskInfo) {
        System.out.println("Launching task " +
            taskInfo.getTaskId().getValue());
        Thread thread = new Thread() {
            @Override
            public void run(){
                //Notify the status as running
                Protos.TaskStatus status = Protos.TaskStatus.newBuilder()
                    .setTaskId(taskInfo.getTaskId())
                    .setState(Protos.TaskState.TASK_RUNNING).build();
                executorDriver.sendStatusUpdate(status);
                System.out.println("Running task " +
                   taskInfo.getTaskId().getValue());
                double pointsUnderCurve=0;
                double totalPoints=0;

                for (double x=xLow; x<=xHigh; x+=(xHigh-xLow)/n) {
                    for (double y=yLow; y<=yHigh; y+=(yHigh-yLow)/n) {
                        double value = expression.evaluate(x);
                        if (value >= y) {
                            pointsUnderCurve++;
```

```
                }
                totalPoints++;
            }
        }
        double area = (xHigh - xLow) * (yHigh - yLow) *
            pointsUnderCurve/totalPoints;
            // Area of Rectangle * fraction of points under curve
        //Notify the status as finish
        status = Protos.TaskStatus.newBuilder()
            .setTaskId(taskInfo.getTaskId())
            .setState(Protos.TaskState.TASK_FINISHED)
            .setData(ByteString
            .copyFrom(Double.toString(area).getBytes())).build();
        executorDriver.sendStatusUpdate(status);
        System.out.println("Finished task " +
            taskInfo.getTaskId().getValue() + " with area : " +
            area);
    }
};

thread.start();
}

@Override
public void killTask(ExecutorDriver executorDriver,
        Protos.TaskID taskID) {
    System.out.println("Killing task " + taskID);
}

@Override
public void frameworkMessage(ExecutorDriver executorDriver,
        byte[] bytes) {
}
@Override
```

```
    public void shutdown(ExecutorDriver executorDriver) {
        System.out.println("Shutting down the executor");
    }

    @Override
    public void error(ExecutorDriver executorDriver, String s) {

    }

    public static void main(String[] args) {
        if (args.length < 6) {
            System.err.println("Usage: MonteCarloExecutor <Expression>
                <xLow> <xHigh> <yLow> <yHigh> <Number of Points>");
        }
        MesosExecutorDriver driver = new MesosExecutorDriver(
            new MonteCarloExecutor(Expression.fromString(args[0]),
                Double.parseDouble(args[1]), Double.parseDouble(args[2]),
                Double.parseDouble(args[3]), Double.parseDouble(args[4]),
                Integer.parseInt(args[5])));
        Protos.Status status = driver.run();
        System.out.println("Driver exited with status "+status);
    }
}
```

大部分回调函数只是简单打印一些表达式，AUC 计算过程发生在 `launchTask()` 中，我们的大部分代码都在其中实现。`launchTask()` 函数需要在一个单独线程中运行，并完成下列任务：

1. 通知 Mesos 将要运行的任务的状态。
2. 运行任务。
3. 任务结束后通知 Mesos 任务的结束状态。

执行器首先向 Mesos 发送状态消息，告知任务正在运行。在蒙特卡洛算法中，点的分布是一致性的而不是随机的，在矩形框内一致性分布大量的点，并且计算表达式在每个点处的值。如果点在曲线下方，那么执行器更新 pointsUnderCurve 计数器的值。曲线所包围的面积可以通过 pointsUnderCurve/totalPoints 乘以矩形框的面积来获得。最后执行器将任务状态更新为 FINISHED 并发送最终结果。

我们利用 MesosExecutorDriver 来管理执行器的生命周期。当参数检查通过后，会通过 run() 创建一个驱动实例并阻塞它。MesosExecutorDriver 接收一个执行器实例作为参数，执行器从命令行参数中解析出表达式和点的坐标。

更新框架调度器

现在我们的执行器已经准备好计算 AUC 了，在调度器中需要创建一个执行器并收集它的结果。下面将在 MonteCarloScheduler 和 App 的构造函数中加入相应的逻辑。

我们加入 taskDone 标志位来确保每个矩形面积只被计算了一次。从启动器的命令行参数中获取表达式和点的坐标，并将这些参数保存下来以便传递给执行器。现在 MonteCarloScheduler 构造函数应该如下所示：

```
public MonteCarloScheduler(String[] args){
    this.args=args;
}
```

App 会将它们传递给 MonteCarloScheduler 实例：

```
MesosSchedulerDriver schedulerDriver = new MesosSchedulerDriver(
    new MonteCarloScheduler(Arrays.copyOfRange(args, 1, args.length)),
        frameworkInfo, args[0]);
```

如果任务没有结束，resourceOffers() 将会接收第一个资源 offer，并利用 MonteCarlo Executor 启动一个新的任务：

```
public void resourceOffers(SchedulerDriver schedulerDriver,
        List<Protos.Offer> offers) {
    if (offers.size()>0 && !taskDone) {
        Protos.Offer offer = offers.get(0);
        Protos.TaskID taskID = Protos.TaskID.newBuilder()
            .setValue("1").build();
        System.out.println("Launching task " + taskID.getValue() +
            " on slave "+offer.getSlaveId().getValue()+" with "+task);
        Protos.ExecutorInfo executor = Protos.ExecutorInfo.newBuilder()
            .setExecutorId(Protos.ExecutorID.newBuilder()
            .setValue("default")).setCommand(EXECUTOR_CMD+args)
            .setName("Test Executor (Java)").build();
        Protos.TaskInfo taskInfo = Protos.TaskInfo.newBuilder()
```

```
                .setName("MonteCarloTask-" +
                taskID.getValue()).setTaskId(taskID)
                .setExecutor(Protos.ExecutorInfo.newBuilder(executor))
                .addResources(Protos.Resource.newBuilder().setName("cpus")
                .setType(Protos.Value.Type.SCALAR)
                .setScalar(Protos.Value.Scalar.newBuilder().setValue(1)))
                .addResources(Protos.Resource.newBuilder().setName("mem")
                .setType(Protos.Value.Type.SCALAR)
                .setScalar(Protos.Value.Scalar.newBuilder().setValue(128)))
                .setSlaveId(offer.getSlaveId()).build();
            schedulerDriver.launchTasks(
                Collections.singletonList(offer.getId()),
                Collections.singletonList(taskInfo));
            taskDone=true;
        }
    }
```

启动任务前还需要检查当前的资源 offer 是否符合任务的需求，但是为了直观，本文省略了这一步骤。

现在执行器已经构造完成，并且已经知道启动 `MonteCarloExecutor` 所需要的命令。为了简化，在这里把 `EXECUTOR_CMD` 设置为 `"java -cp MonteCarloArea.jar:mesos-0.20.1-shaded-protobuf.jar -Djava.library.path=./libs org.packt.mesos.MonteCarloExecutor"`，但是正常情况下应该从外部环境变量中来读取这个值。

`Task` 中包含了任务的资源需求，指定了需要使用哪种类型的执行器，它还通过所接收的资源 offer 中的 `slaveId` 指定了在哪里启动任务。这里，我们使用 1 个 CPU 和 128 MB 内存来启动任务。最终驱动中的 `launchTasks()` 方法被调用，我们也可以在调用 `launchTasks()` 的同时指定过滤器来规定对剩余资源 offer 的限制。`launchTasks` 接收一个包含 `OfferId` 和 `TaskInfo` 的集合作为参数，因此我们创建了一个只有元素的列表。接受 `OfferId` 和 `TaskInfo` 实例作为参数的方法现在已经被废弃了。

`statusUpdate()` 函数检查 `TASK_FINISHED` 状态是否被更新，如果接收到更新，则打印执行器所发送的 AUC 值：

```
public void statusUpdate(SchedulerDriver schedulerDriver,
        Protos.TaskStatus taskStatus) {
    System.out.println("Status update: task " +
```

```
        taskStatus.getTaskId().getValue() + " state is " +
        taskStatus.getState());
    if (taskStatus.getState().equals(Protos.TaskState.TASK_FINISHED)) {
        double area = Double.parseDouble(taskStatus.getData()
                                .toStringUtf8());
        System.out.println("Task " + taskStatus.getTaskId().getValue() +
            " finished with area : " + area);
        schedulerDriver.stop();
    } else {
        System.out.println("Task " + taskStatus.getTaskId().getValue() +
            " has message " + taskStatus.getMessage());
    }
}
```

现在来更新框架，需要提供曲线及矩形框的坐标信息：

```
ubuntu@master:~ $ java -cp MonteCarloArea.jar:mesos-0.20.1.jar:protobuf-2
.5.0.jar -Djava.library.path=libs/ org.packt.mesos.App "zk://master:2181/
mesos" "x" 0 10 0 10 100
```

这个命令会运行框架并利用 100 个点来计算面积，即曲线 y=x 与由 (0,0)、(10,0)、(0,10) 和 (10,10) 所定义的矩形框所包围的面积。框架的更新过程会在框架注册后打印出和下面类似的信息：

```
...
Launching task 1 on slave 20141220-113457-184623020-5050-1249-0 with " x"
 0.0 10.0 0.0 10.0 100
Status update: task 1 state is TASK_RUNNING
Task 1 has message
Status update: task 1 state is TASK_FINISHED
Task 1 finished with area : 50.495049504950494
...
```

该框架计算的 AUC 值为 50.495049504950494，和实际的 50 十分接近。还可以使用更多的点使得计算结果更加准确。此外，在 slave 中可以查看诸如分配任务、slave 启动容器、slave 获取执行器和启动任务的各种事件日志；还可以看到监控及状态更新的信息，由于本书篇幅所限，这里不做过多讨论。

运行多个执行器

目前为止所有的计算都是在一个执行器上完成的。Mesos 作为分布式框架的真正优势是可以同时启动多个任务来完成一项工作，但是我们还没有使用到。Mesos 的 API 可以帮助我们十分轻松地将任务从一个执行器扩展到多个执行器。由于 MonteCarloExecutor 已经可以计算指定矩形框下的曲线面积，所以可以很容易地对其扩展，以便使用更多的计算资源。将整个矩形框分割成多个小矩形框，然后要求 MonteCarloExecutor 计算每个小矩形框中的曲线面积。现在调度器需要负责将整个工作切分成多个小任务。在这个场景中，MonteCarloScheduler 需要完成以下任务。

1. 将指定矩形框切分成多个小矩形框，这个工作会在 MonteCarloScheduler 的构造函数中完成。任务所需参数存放在 task 域，tasksSubmitted 和 tasksCompleted 计数器用来记录多少任务正在运行，以及多少任务已经完成，totalArea 记录从所有结束任务中汇总的 AUC 值：

```
public class MonteCarloScheduler implements Scheduler {
    private LinkedList<String> tasks;
    private int numTasks;
    private int tasksSubmitted;
    private int tasksCompleted;
    private double totalArea;

    public MonteCarloScheduler(String[] args, int numTasks) {
        this.numTasks=numTasks;
        tasks=new LinkedList<String>();
        ...
        // code for field assignments
        ...
        double xStep=(xHigh-xLow)/(numTasks/2);
        double yStep=(yHigh-yLow)/(numTasks/2);

        for (double x=xLow; x<xHigh; x+=xStep){
            for (double y=yLow; y<yHigh; y+=yStep) {
                tasks.add(" \" " + args[0] + " \" " + x + " " +
                    (x + xStep) + " " + y + " " + (y + yStep) +
                    " " + args[5]);
            }
```

```
        }
```

2. 如果有任务等待执行，则将它们分发到执行器中，传入合适的矩形框坐标，并利用tasksSubmitted计数器来设置TaskId和ExecutorId：

```
for (Protos.Offer offer : offers) {
    if(tasks.size()>0) {
        // launch a task
    }
}
```

3. 任务结束后收集并汇总结果。一旦所有任务结束，则打印最后的面积计算结果并关闭驱动：

```
if (taskStatus.getState().equals(Protos.TaskState.TASK_FINISHED)){
    tasksCompleted++;
    double area = Double.parseDouble(
                                taskStatus.getData().toStringUtf8());
    totalArea+=area;
    System.out.println("Task" + taskStatus.getTaskId().getValue() +
        " finished with area : " + area);
} else {
    System.out.println("Task " + taskStatus.getTaskId().getValue() +
        " has message " + taskStatus.getMessage());
}
if (tasksCompleted==numTasks){
    System.out.println("Total Area : " + totalArea);
    schedulerDriver.stop();
}
```

在命令行参数中需要新加入一个参数来指明启动几个任务：

```
MesosSchedulerDriver schedulerDriver = new MesosSchedulerDriver(
    new MonteCarloScheduler(Arrays.copyOfRange(args,2,args.length),
        Integer.parseInt(args[1])), frameworkInfo,args[0]);
```

在做了如上改动后可以再次启动框架。现在利用四个任务来计算同样的曲线 $x = y$ 及同样的矩形框所包围的面积：

```
ubuntu@master:~ $ java -cp MonteCarloArea.jar:mesos-0.20.1.jar:protobuf-2
```

```
.5.0.jar -Djava.library.path=libs/ org.packt.mesos.App "zk://master:2181/
mesos" 4 "x" 0 10 0 10 100
```

我们将看到和下面类似的输出结果：

```
Scheduler registered with id 20141220-113457-184623020-5050-1249-0004
Launching task 1 on slave 20141220-113457-184623020-5050-1249-1 with " x"
 0.0 5.0 0.0 5.0 100
Launching task 2 on slave 20141220-113457-184623020-5050-1249-0 with " x"
 0.0 5.0 5.0 10.0 100
Status update: task 2 state is TASK_RUNNING
Task 2 has message
Status update: task 2 state is TASK_FINISHED
Task 2 finished with area : 0.0
Launching task 3 on slave 20141220-113457-184623020-5050-1249-0 with " x"
 5.0 10.0 0.0 5.0 100
Status update: task 1 state is TASK_RUNNING
Task 1 has message
Status update: task 1 state is TASK_FINISHED
Task 1 finished with area : 12.623762376237623
Status update: task 3 state is TASK_RUNNING
Task 3 has message
Status update: task 3 state is TASK_FINISHED
Task 3 finished with area : 25.0
Launching task 4 on slave 20141220-113457-184623020-5050-1249-1 with " x"
 5.0 10.0 5.0 10.0 100
Status update: task 4 state is TASK_RUNNING
Task 4 has message
Status update: task 4 state is TASK_FINISHED
Task 4 finished with area : 12.625
Total Area : 50.24876237623762
I1220 13:50:03.771772 1733 sched.cpp:747] Stopping framework
'20141220-113457-184623020-5050-1249-0004'
```

可以观察到，调度器将大矩形框切分成如下四个小矩形框并分配给不同的任务：

- (0,0),(5,0),(5,0),(5,5)

- (0,5),(5,5),(0,10),(5,10)
- (5,0),(10,0),(5,5),(10,5)
- (5,5),(10,5),(5,10),(10,10)

四个矩形框的 AUC 分别为 `12.62`、`0`、`25` 和 `12.62`，总计的结果为 `50.24`。这个值比使用一个任务执行器所得出的 `50.49` 更加接近真实值 50。这里我们需要将框架运行在一个由两台主机所组成的小型集群上，所以会看到任务 3 和 4 需要等待任务 1 和 2 结束后才能运行。还可以尝试更大数量的任务切分，可能会得到更加精确的结果。

高级主题

上文已经介绍了如何实现一个非常简单的 Mesos 框架，真实世界的框架远比其复杂。本节简要讨论一些高级主题和一些注意事项。

一致性调解

Mesos 做了大量工作来隐藏构建分布式应用的细节问题并提供了简单易用的 API。尽管 Mesos 为框架开发者提供了可靠的通信机制，但是还是要注意框架本身是一个分布式系统这件事情。Mesos 使用一种 actor 编程模型进行消息传递，并且混用了*至多一次*和*至少一次*两种语义。框架和 master 之间的消息传递可能失败和丢失。一旦发生这种情况，框架和 master 各自观察到的状态将会出现不一致。一个可能的场景是框架向 master 发送了一个消息，master 在接收到消息前发生故障，或者在将消息发送给 slave 前发生故障。此时任务认为 master 已经收到响应的消息，而 master 本身其实并不知道。为了处理这种问题，Mesos 中存在着一些机制来保证 master 和框架之间的信息一致。

Mesos 提供了一致性调解的 API（`http://mesos.apache.org/documentation/latest/reconciliation/`）使得框架可以在发现不一致情况的故障时采取合适的应对措施。故障检测由 master 和调度器驱动在连接断开和重注册时执行。有两种类型的一致性调解：

- offer 一致性调解
- 任务一致性调解

由于 offer 的状态不会被持久化，断开连接后 offer 会失效，重新注册时会被重新生成，因此一致性调解可以自动执行。由于调度器驱动不会持久化任何信息，任务一致性调解必须由框架显式地在故障发生时执行。在向 Mesos master 重新注册的时候，框架必须执行任务一致性调解。框架将其认为仍在执行的所有任务的 TaskStatus 以列表形式发送给 master。收到这个消息后，master 检查 `TaskStatus` 中的 `TaskID` 和 `slaveID` 域并送回列表中每个

任务的任务状态。如果框架发送的列表为空，那么 master 将返回它所知的所有任务的状态信息，那些无法找到的任务状态会被更新为 TASK_LOST。框架的编写者需要对所有状态有争议的任务进行调解并利用和下面类似的算法类更新每个任务的状态：

```
startTime = currentTime();
remainingTasks = { All non-terminal tasks }
While(remainingTasks <> {}){
    reconcile(remainingTasks)
    wait for update with timeout truncated exponential backoff;
    remainingTasks={all tasks in remainingTasks that has not been updated
        since startTime}
}
```

这里我们循环等待，直到 remainingTasks 列表不为空，然后针对 remainingTasks 中的任务进行一致性调解。对于每个接收到的状态更新，remainingTasks 中对应的任务也会进行更新。如前所述，一致性调解算法需要在每次重新注册时执行。框架需要保证在指定时间内只能有一个一致性调解在执行。这在未来可以由客户端的语言库来实现，这样一致性调解的过程对于框架的编写者就是完全透明的。

有状态应用

目前 Mesos 的设计中，一旦任务结束，资源就会被 Mesos 回收，并且不保证任务在重启后还可以在相同的 slave 上获得资源。这使得对于需要持久化存储（如数据库或者分布式文件系统的框架）很难在 Mesos 上运行。目前有很多人在致力于扩展 Mesos，使得在 Mesos 上可以更加方便地开发有状态应用。目前，需要持久化磁盘的框架可以使用一个未被管理的路径（如 var/lib/cassandra）写入数据。如果多个框架使用相同的路径写入数据，所产生的行为是不可预测的。因此在更多关于有状态应用的支持（https://issues.apache.org/jira/browse/MESOS-1554）加入 Mesos 之前，需要保证两个框架不会争夺同一个路径。

开发者资料

作为一个开发者，参考现有的 Mesos 内建框架和其他的资料对于开发 Mesos 框架十分有帮助。

框架设计模式

目前还处在大量 Mesos 框架不断出现的早期阶段，但是已经可以从不同的 Mesos 框架分配调度使用资源的方式中总结出以下几种模式。

- 资源协调者：这些框架只是作为其他框架的资源协调者。本书第 5 章中所见到的服务框架都是这样的元框架。这类框架十分普遍，通常它们对需要进行资源调解的应用的了解有限。
- 基于负载：这类框架根据框架的负载来调节资源的使用。Marathon 和 Aurora 这两种可以根据约定自动进行扩容和缩容的框架就是这类框架的例子。持续集成框架也是这类框架的例子之一，Jenkins on Mesos 和 Gitlab CI on Mesos 框架可以根据等待队列的长度来调节资源的使用。一些机会主义的框架也属于这类框架。
- 基于预留的框架。这类框架在运行时静态预留资源，需要在执行前获得所有资源，并且在框架的整个生命周期内都持有这些资源。本书第 1 部分的第 2 章～第 4 章中所介绍的例子都属于这类框架。通过这种模式，现有的分布式框架（例如已有的分布式存储系统）可以很轻松地移植到 Mesos 框架中。

值得一提的是框架模式并不只有这几种，每个框架也不是只能遵循一种模式。一些框架混合了多种设计模式，另一些框架不符合上述所说的任何一种模式。这也是 Mesos 的一个主要优势，它提供了基础设施而不限制使用方式，允许各种资源分配策略的实现。

框架测试

分布式框架的测试是一件十分有挑战的事情。由于隐藏了分布式消息传递的复杂细节，Mesos 使得测试问题变得简单了一些。对于大多数框架，保证执行器的准确性是测试的主要目标。执行器的测试依赖于执行器的逻辑。在上文的例子中，可以针对不同的曲线编写单元测试并比较输出结果来进行测试。调度器的测试主要确保调度器可以正常地进行隔离。在上文的例子中，需要编写测试用例来确保不同任务的切分结果正确。框架测试最困难的部分在于需要确保测试的框架可以正确地和现有框架共存。一种方法是截获 Mesos API 请求来验证不同的回调完成后调度器的状态是否正确。不过大部分框架不需要这种级别的测试。

RENDLER

Mesosphere 创建了一个 Mesos 网页爬虫的框架 RENDLER (`https://github.com/mesosphere/RENDLER`)，这是 Mesos 开发者学习框架开发的很好的切入点。RENDLER 提供了

Go、Python、Scala、C++ 和 Java 语言版本的实现，并且提供了虚拟机的运行环境来试验爬虫框架。

Akka-mesos

Akka-mesos（`https://github.com/drexin/akka-mesos`）提供了使用 Akka 库来构建 Mesos 框架的方法。Akka（`http://akka.io`）是一个利用 actor 编程模型来化简 reactive 应用开发的工具链。Akka-mesos 提供了构建 Mesos 框架通用方法和模式的帮助结构。

小结

本章浏览了 Mesos API 并实现了一个计算不同曲线域矩形框所包围面积的框架，还介绍了一些高级主题，以及在开发 Mesos 框架中十分有用的开发资料。下一章将会看到在管理 Mesos 集群中所遇到的挑战。

第8章 管理 Mesos

本章内容面向的主要是管理员和开发者，将介绍一系列 Mesos 集群的部署和管理工具，并介绍在 Mesos 部署过程中实现多租户和高可用时所不可或缺的特性；此外还会探索 Mesos REST 接口及 CLI 的使用方式。

本章包含如下主题：

- 部署
- 升级
- 监控
- 多租户
- 高可用
- 维护状态
- Mesos 接口

部署

Mesos提供了在多种环境下运行 Mesos 集群的便利性。部署 Mesos 不需要任何特殊的硬件，可以在公有云和私有云上进行部署，也可以在多个操作系统平台上运行。因此只需选择适合需求的硬件设备，Mesos 就可以运行在其上并能高效地解决商业问题。

Mesos 提供了 shell 脚本来辅助集群的部署。如同在第 1 章中所提到的，`mesos-start-cluster.sh` 和 `mesos-stop-cluster.sh` 脚本可以帮助启动和停止 slave 和 master 进程。

手动安装不仅烦琐而且十分容易出错，因此脚本是一种更为便捷地使用 Mesos 的方式。如果要在生产环境中运行 Mesos，那么强烈推荐使用自动化部署工具。

第 1 章中讲述了如何在 AWS 上创建 Mesos 服务。社区中也有人开发了 AWS CloudFormation 模板。Mesosphere 中提供了在 GCE 和 DitigalOcean 上安装 Mesos 的详细步骤（`http://mesosphere.com/docs/getting-started/cloud/`）。Mesosphere 还提供了各个操作系统平台下的二进制打包文件。当然也可以手工在各个云平台上安装和管理 Mesos。

Mesos 提供了很多自动化部署集群的选项，它能够和运维人员多年来一直使用的其他部署工具很好地集成。社区中有着大量为现有主流工具部署 Mesos 的支持和插件，例如 Chef cookbook、Puppet modules 及 Ansible playbook，可以在 `http://mesos.apache.org/documentation/latest/tools/` 和 `https://github.com/dharmeshkakadia/awesome-mesos` 找到更多相关的支持。

升级

Mesos 的升级十分频繁，升级步骤相对也很简单：

- 安装新的 master 二进制包。
- 重启 master 进程。
- 安装新的 slave 二进制包。
- 重启 slave 进程。
- 升级调度器并链接新的 Mesos 库。
- 重启调度器。

如果需要，执行器也可以通过和新的 Mesos 库链接的方式实现升级。如同第 6 章中所介绍的，任务可以在 slave 进程和执行器重启的过程中存活。如果某一版本的 Mesos 需要升级底层操作系统的内核，那么可能需要重启 master 和 slave 所在的机器。Mesos 的升级文档（`http://mesos.apache.org/documentation/latest/upgrades/`）提供了更多有关不同版本升级的信息。

在维护 Mesos 集群的过程中还需要维护不同的框架，保证它们的版本能够及时更新。对于一些框架如 Spark，它的调度器和执行器由上游的 Spark 项目维护。大部分移植到 Mesos 的框架都是在不同于 Mesos 的项目中被维护。长期来看，这使得在生产环境中维护这些框架变得十分困难。Mesos 作为数据中心的操作系统需要一个和 yum、apt 或者 brew 类似的包管理软件，这样用户可以描述他们需要安装哪些框架，并由包管理器来负责安装。在本书写作时，关于框架包管理器的讨论正在进行中。Mesosphere 所开发的 Universe（`https://github.com/mesosphere/universe`）是一个包仓库的例子。

监控

监控是保证集群正常运行的重要工作。Mesos 和现有的监控解决方案可以很好地集成，并提供了插件来支持 Nagios、collectd（https://github.com/rayrod2030/collectd-mesos）等监控系统。这使得运维人员可以轻松地利用他们多年来在现有工具上的经验来管理 Mesos。插件的安装步骤和每个工具的特点相关，大多数插件安装的步骤类似，并不需要 Mesos 方面的专门知识。监控系统可以通过 HTTP 接入点获取监控信息。http:///metrics/snapshot 提供了关于资源使用的详细报告，如下面的代码所示：

```
{
    "master/cpus_percent":0,
    "master/cpus_total":2,
    "master/cpus_used":0,
    "master/disk_percent":0,
    "master/disk_total":32808,
    ...
    "registrar/queued_operations":0,
    "registrar/registry_size_bytes":405,
    "registrar/state_fetch_ms":8.523776,
    "registrar/state_store_ms":12.186112,
    ...
    "system/cpus_total":2,
    "system/load_15min":0.12,
    "system/load_1min":0.22,
    "system/load_5min":0.24,
    "system/mem_free_bytes":1739386880,
    "system/mem_total_bytes":2099142656
}
```

Mesos 社区积极地为运维人员开发了许多工具。Angstrom（https://github.com/nqn/angstrom）是其中一个用来收集 Mesos 的各种指标并提供集群全局状态的系统。它由 Go 语言编写，并且包含了网页界面及 REST API。通过 API 还可以获取一段时间内的各种统计信息及 CPU、内存和磁盘的汇总信息。

容器网络监控

从 0.20.0 版本开始，Mesos 支持对每个容器网络进行监控（http://mesos.apache.org/

documentation/latest/network-monitoring)。它不需要依赖于 slave 上运行的任务来提供网络统计数据，它会在 slave 的 /monitor/statistics.json 中提供获取数据的接入点。

网络监控是最近新引入的特性，并且只能在 Linux 内核版本高于 3.6 的系统上运行。这个特性依赖于 Linux 内核中的 network namespace 以及 libnl3 和 iproute 包。网络监控默认是关闭的，必须通过 --with-network-isolator 选项来开启，并通过下面的方式来构建 Mesos：

```
ubuntu@master:~/mesos/build $ ../configure --with-network-isolator ; make
```

当网络监控开启后，每个容器都会获得主机上的端口的一个子集。默认情况下，Linux 的临时端口范围为 32768～61000。由于临时端口从主机直接映射到容器，为了能让容器在主机上安全运行，Mesos 进一步限制了端口范围。下面是一个将端口进行限制的配置样例：

```
ubuntu@master:~ $ echo "57345 61000" > /proc/sys/net/ipv4/ip_local_port_
range
```

上面的变化在主机重启后才会生效，现在容器可以使用端口号为 32768～57344 的端口。当启动 slave 时，需要在 --isolation 中加入 network/port_mapping，在 --resources 中加入 ephemeral_ports，还要设置其他一些选项：

```
ubuntu@master:~ mesos/build/bin $ mesos-slave \
--isolation=cgroups/cpu,cgroups/mem,network/port_mapping \
--resources=cpus:22;mem:62189;ports:[31000-32000];disk:400000;ephemeral_
ports:[32768-57344] \
--ephemeral_ports_per_container=1024\
#rest of the options
```

--ephemeral_ports_per_container 还指定了给每个容器分配多少随机端口，在之前的例子中，我们为每个容器分配了1024 个端口。出于性能的考量，这个值最好是 2 的整数次方。容器的总数量也会受端口数目的限制，每个机器所能拥有的容器数量最多为总端口数除以每个容器端口数的值这么多。Mesos 通过 ports 资源向调度器暴露非临时端口，并由调度器对它们进行分配。当这些设置完毕，就可以通过 /monitor/statistics.json 接入点来获取如下统计信息：

- net_rx_bytes
- net_rx_dropped
- net_rx_errors

- net_rx_packets
- net_tx_bytes
- net_tx_dropped
- net_tx_errors
- net_tx_packets

从 0.21 版本新引入的容器网络栈使得我们可以控制容器网络的网络带宽，以避免某个容器消耗大量带宽资源而对其他容器产生影响的现象。这需要开启 `--egress_rate_limit_per_container` 选项，并指定希望的速度限制，例如 `--egress_rate_limit_per_container=500KB`。

多租户

对于数据中心内核来说，能够让多个用户来共享集群资源和框架是必不可少的一个功能。如果没有合适的多租户支持，不同的用户和组织就需要静态划分集群来保证隔离，这也就丧失了 Mesos 所提供的优点。Mesos 以最大的粒度来支持多租户，这一点将在下文介绍。

授权和鉴权

鉴权是 Mesos 中的一个重要功能。Mesos 从 0.20.0 版本（`http://mesos.apache.org/documentation/latest/authorization/`）开始支持框架授权。它使用 **SASL 库**来实现鉴权，并支持多种鉴权机制。它包含：

- 框架以授权的角色形式进行注册和重新注册。
- 框架以授权用户的身份启动任务。
- 授权过的管理员利用 `shutdown/` 接入点和 HTTP API 来关闭框架。

Mesos 通过**访问控制列表（ACL）**来实现授权。对于每个和 Mesos master 的交互，master 都会检查当前的请求是否经过 ACL 授权。如果没有经过授权则返回一个错误消息。每个 ACL 条目由目标、动作和对象三部分组成，可以这样来帮助记忆，“目标”可以在“对象”上执行“动作”。

1. 动作——目前 Mesos ACL 中支持如下动作：
   - `register_framewors`，该动作可以注册或者重新注册框架。当一个框架试图向 master 进行注册或者重新注册时，会在 `FrameworkInfo` 中填入相应的 ACL 信息。master 会去检查 `FrameworkInfo.principal` 是否授权

FrameworkInfo.role 接收资源 offer。

- run_tasks，该动作可以运行执行器或任务。ACL 检查当前用户是否被 FrameworkInfo.principal 授权运行执行器或者任务。
- shutdown_frameworks，该动作可以关闭框架。ACL 检查当前用户是否被 FrameworkInfo.principal 授权关闭框架。

2. 目标——当前只支持 principals。principals 可以是在 register_frameworks 或者 run_tasks 动作中所指定的框架管理员，或者是 shutdown_frameworks 动作中的用户名。
3. 对象——当前支持的对象类型有：
   - roles，框架可以利用 register_frameworks 动作通过指定角色来分配资源。
   - users，用户名将会在 run_task 动作开启任务或者执行器的时候使用。
   - framework_principals，框架管理员可以通过 HTTP 接入点来关闭框架。

ACL 规则是按顺序进行匹配的，这意味着第一条匹配的 ACL 规则将决定这个请求是否被授权。如果没有 ACL 规则和当前请求匹配，则 ACLs.permissive 将决定当前请求是否被授权。如果该选项被设置为 true，则未被匹配的请求将被授权。默认情况下 ACLs.permissive 被设置为 true。ACL 的配置为一个 JSON 文件，描述了所有的规则，下面是一个 ACL 文件示例：

```
{
  "register_frameworks": [
    {
      "principals": { "values": ["foo"] },
      "roles": { "values": ["developer","operation"] }
    },
    {
      "principals": { "values": "bar" },
      "roles": { "values": ["barOnlyRole"] }
    },
    {
      "principals": { "type": "NONE" },
      "roles": { "values": ["barOnlyRole "] }
    }
  ],
```

```
  "run_tasks": [
    {
      "principals": { "values": ["a", "b"] },
      "users": { "values": ["abUser"] }
    },
    {
      "principals": { "values": [ "c" ] },
      "users": { "values": ["cUser"] }
    },
    {
      "principals": { "values": [ "c" ] },
      "users": { "type": "NONE" }
    },
    {
      "principals": { "type": "ANY" },
      "users": { "values": ["guest"] }
    },
    {
      "principals": { "type": "NONE" },
      "users": { "values": ["root"] }
    }
  ],

  "shutdown_frameworks": [
    {
      "principals": { "values": ["a", "b"] },
      "framework_principals": { "values": ["c"] }
    },
  ]
}
```

在上面的配置文件中，foo 框架可以以开发人员和运维人员的角色进行注册，管理员 bar 可以以 barOnlyRole 的角色注册，而且其他人都不得以 barOnlyRole 的角色注册。框架 a 和 b 可以以 abUser 的身份来运行，框架 c 只能以用户 cUser 的身份运行。任何框架都可以以 guset 身份运行任务，但是都不能以 root 身份运行任务。

下面是一个简单的例子：

```
{
  "permissive": "false",
  "register_frameworks": [
    {
      "principals": { "values": ["foo"] },
      "roles": { "values": ["fooOnlyRole"] }
    },
  ],
  "shutdown_frameworks": [
    {
      "principals": { "values": ["ops"] },
      "framework_principals": { "type": "ANY" }
    }
  ]
}
```

上例展示了 `permissive` 设置为 `false` 的情况。这种情况下，管理员 `foo` 只能以 `fooOnlyRole` 的角色进行注册。其他框架也不能以任何其他用户的身份进行注册。只有管理员 `ops` 可以利用 HTTP 接入点关闭框架。

slave 注册的鉴权、Kerberos 集成、网络信息加密等其他高级功能都在 Mesos 之后发行版的规划之中。在本书写作时，Mesos 中所有的网络通信都是未加密的，在某些环境中这可能是个潜在的隐患。目前，将所有消息通过 SSL/TLS 传输的工作正在进行中（`https://issues.apache.org/jira/browse/MESOS-910`）。

如果希望允许多个 UNIX 用户向同一集群提交任务，需要以 root 形式运行 Mesos slave，否则 slave 在 setuid 时会失败。

API 速率限制

Mesos 的多框架模式使得集群资源得到更加充分的利用，但是如果一个框架过于贪婪地使用资源，就会造成其他框架性能的下降。不同框架有着不同的**服务等级（SLA）**和不同的处理特性（批处理、在线服务等）。因此，一个框架可以一直不断地让 master 只处理自己的请求，而阻止 master 处理其他框架的请求，这会造成其他框架的饥饿。为了避免这种情况，Mesos 支持框架 API 速率限制的功能（`http://mesos.apache.org/documentation/`

latest/framework-rate-limiting)。运维人员可以指定每个 Mesos 每秒可以处理的框架请求的数量，超出数量限制的消息会被存储在 Mesos master 的内存中。

当一个框架超出它的容量限制后，框架将会接收到 FrameworkErrorMessage 消息。这将会导致调度器驱动异常退出并调用 error 回调函数，这一步并不会杀死任何任务和调度器本身。如果框架认为 master 并没有处理自己的全部请求，框架可以选择重启或者故障恢复来继续工作。

框架的速率限制配置文件是一个 JSON 格式的文件，它有下面几个参数。

- principal：用来指定哪些框架需要被限制。
- qps：每秒请求的个数。
- capacity：控制 Mesos master 内存队列中的指定 principal 等待消息个数。如果 qps 没有指定，那么会忽略 capacity 的值。如果 qps 被指定，而 capacity 没有被指定，那么默认 capacity 的值是无限大。

速率限制可以通过在 master 启动时利用 --rate-limits 指定一个配置文件的方式来开启。配置文件在 master 启动时被加载生效直到 master 结束退出。下面是一个配置文件的样例：

```
{
  "limits": [
    {
      "principal": "a",
      "qps": 100,
      "capacity": 100000
    },
    {
      "principal": "b",
      "qps": 500
    },
    {
      "principal": "c",
    }
  ],
  "aggregate_default_qps": 5000,
  "aggregate_default_capacity": 1000000
}
```

这里 master 对 a、b、c 三个框架定义了限制，其余的框架共享每秒 5000 个请求和

100000 条等待消息的限制。框架 a 的 qps 被限制为 100，允许有 100000 条等待消息；框架 b 的 qps 被限制为 500，等待消息的数量没有限制；框架 c 没有任何限制。aggregate_default_qps 和 aggregate_default_capacity 用来控制那些没有明确限制的框架的 qps 和 capacity。如果这些选项没有在配置中提及，那么剩下的框架都不会被限制。

capacity 值的设置需要十分谨慎。如果 capacity 值过低，那么资源将得不到充分利用；如果 capacity 值过高，则队列信息可能会导致 master 内存不足。master 可以处理的消息数目和 master 框架的内存限制以及消息的大小相关。Mesosaurus（https://github.com/mesosphere/mesosaurus）是 Mesos 的一个评测框架，它可以在集群中模拟不同的工作负载和不同的框架行为。在本书写作时，该工具还在开发中，不过已经有了许多实用的功能。我们可以先将 capacity 设置成一个较低的值，以应对框架消息的暴涨，并在需要时适当增加该值。

Mesos master 通过 metrics HTTP 接入点向外提供每个框架接收和处理的消息个数。http://<master>/metrics/framework/frameworkA/messages_received 和 http://<master>/metrics/framework/frameworkA/messages_processed 给出了 master 从 frameworkA 接收和处理的消息个数。框架可以监控这些数据来了解当前框架的队列长度。正常情况下，这两个数字的差距不会太大，当框架超出使用限制时，这两个值的差距会变得十分大。

尽管速率限制这个功能十分有用，但是在本书写作时，这个功能还不完善，在生产环境要谨慎使用。现在社区中还有这样的提议，即允许框架在接收到 master 发来的软警告后，自己降低发送消息的速率。这样框架调度器将有机会把自己控制在限制范围以内，而不会抛出错误信息。

高可用

大多数的商务业务的持续运行都需要建立在基础设施的高可用之上。作为数据中心内核，Mesos 不仅提供了高可用的保证，其自身特点也使得多个 Mesos 组件可以迅速从故障中恢复。

master 高可用

Mesos 的高可用主要依赖于 Mesos master 的可用性。如之前所介绍的，Mesos 使用 ZooKeeper 来确保至少一个 master 可以提供服务，即使发生故障也是如此。ZooKeeper 选举出一个主 master，其他的 master 作为守护。在第 6 章中介绍过如何利用 ZooKeeper 运

行 Mesos。

在对 ZooKeeper 进行设置时至少需要配置 dataDir。该值必须被设置为指向持久存储的目录位置，不要使用 ZooKeeper 样例配置中默认配置的 /tmp。在多机部署的情况下，ZooKeeper 从 myid 文件中读取自己的身份信息。该文件在 dataDir 下，并且只包含一个独特的机器标示（例如 1）。ZooKeeper 配置项的具体细节可以参考 ZooKeeper 的文档（https://zookeeper.apache.org/）。该网站上除了有管理文档外，还有 Flavio Junqueira 和 Benjamin Reed 编写的 *ZooKeeper: Distributed Process Coordination*，里面介绍了许多生产环境使用 ZooKeeper 的最佳实践。

最好将 Mesos 用来进行主选举的 ZooKeeper 集群和其他系统所需要的 ZooKeeper 集群隔离开，以避免其他系统的不当使用导致 ZooKeeper 宕机而影响 Mesos 集群。

Mesos master 进程在失去和 ZooKeeper 的连接后自杀。在有 $2N-1$ 个 master 的情况下，Mesos 可以在 $N-1$ 个 master 发生故障的情况下正常工作。为了保证高可用，推荐使用 3 或 5 个 master，这可以容忍 1 或 2 个 master 发生故障的情况。Mesos 的个数必须保证是 $2N-1$ 个，运行更多的 master 进程可能会导致 replicated log 的损坏。这一点可以通过外部工具或者建立 Mesos 白名单（https://issues.apache.org/jira/browse/MESOS-1546）来做限制。如果条件允许，不同的 ZooKeeper 集群和不同的 Mesos master 集群应该在相互独立的故障域运行。

运维人员可能会根据 Mesos 集群内机器数量的增加或者减少来调整 master 的个数。当 master 数量变化时，要注意清空 replicated log。目前更改 master 的数目必须重新启动集群。可以参考在线的配置来调整 master 的数目 https://issues.apache.org/jira/browse/MESOS-683。下面这个例子介绍了将 master 的个数从 3（A、B、C）变为 5（A、B、C、D、E）的步骤。

1. 停止当前启动参数为 --quorum=2 的 A、B、C 三个 master 节点。
2. 将 quorum 变为 3，启动参数调整为 --quorum=3，并重启之前的 master A、B、C。
3. 以 --quorum=3、空日志的条件依次启动新的 master 节点 D、E，并让它们和现有 replicated log 同步。

缩减 master 数目的过程和上面类似，在 Mesos 操作指南中有详细的步骤（http://mesos.apache.org/documentation/latest/operational-guide）。

需要注意 Mesos master 不能够自动重启，需要手动或者借助外部管理工具 runit（http://smarden.org/runit）和 monit（https://mmonit.com/monit）来重启。Mesos 中有很多细小的功能来避免代码缺陷或者运维失误导致集群无法提供服务的情况。例如：

- Mesos master 在失去和 ZooKeeper 的连接后自杀。这样 master 不会在故去的状态上处理请求。
- Mesos slave 忽略非主 master 发送的关闭请求。
- 限制 slave 移除速率，我们将在下面进行介绍。

限制 slave 移除速率

Mesos 支持对 slave 移除的速率进行限制。例如，在一个假设的场景下，一个实现有缺陷的 master 进程丢失了所有 slave 的确认消息，并开始关闭 slave。如果不做速率限制，那么 Mesos 在几秒内将失去所有 slave。通过引入速率限制，slave 的移除速度会变慢，使得运维人员有机会对这一异常现象进行处理。在这种场景下，运维人员可能通过监控发现 slave 数量不断下降，并通过杀掉有缺陷的 master 进程避免整个集群无法提供服务。Mesos 支持下面两种类型的 slave 移除速度限制。

- 通过 `--recovery_slave_removal_limit` 控制 slave 在故障恢复时被移除的比例。这个选项指定了可以从 registry 中所移除的 slave 的最大比例。举例来说，假设这个值是 10%，那么在故障恢复的过程中，如果 master 不能和 10% 以上的 slave 连接，那么 master 将自杀而不会继续移除 slave。该值默认为 100%，也就意味着不会对故障恢复时 slave 的移除做限制。
- 通过 `--slave_removal_rate_limit` 可以控制 slave 移除的速度。这个选项规定了在一段时间范围内所移除 slave 的最大个数。例如，该值为 3/40 minutes 意味着在 40 分钟的时间窗口内最多只能移除两个 slave。默认情况下，slave 会在健康检查失败后立即移除。速率限制功能在 Mesos 0.22 版本被引入。

slave 恢复

Mesos 支持 slave 的无缝升级（`http://mesos.apache.org/documentation/latest/slave-recovery/`）。能够在 slave 升级的同时不影响现有任务的运行十分重要，这可以在不影响现有应用运行的情况下将 slave 升级至最新版本。执行器可以在 slave 进程不存在的情况下继续工作，并在 slave 重启后与之重连。该机制也为 slave 的故障恢复提供了保障，这对于那些需要长时间启动的有状态服务来说十分重要，故障恢复的方式对服务的可用性有着深远的影响。

slave 的恢复是通过将任务和执行器的信息保存到硬盘创建检查点的方式实现的。Mesos slave 在重启时可以通过检查点的信息重新和执行器建立连接。slave 恢复只有在检查点开启，并且框架指明需要恢复的情况下才会执行。开启检查点的框架只会从开启检查点的

slave 上获取资源 offer。框架可能会由于某些原因而关闭检查点，例如 IO 性能损耗的考量。

理论上讲可以在单个节点上运行多个 slave 进程，但并不推荐这样做。一个 slave 进程就可以管理一个节点上所有的资源，增加进程只会增加无用的性能开销。

默认情况下检查点功能是关闭的，开启检查点功能需要在 slave 启动时指明 `--checkpoint` 选项。`--recover` 选项可以规定 slave 是重新和运行的执行器相连（`--recover=reconnect`）还是将执行器杀掉（`--recover=cleanup`）。如果没有检查点信息，那么 slave 将以一个新的 slave 身份向 master注册。恢复默认会在15分钟后终止。如果恢复的过程需要更长时间，则可以通过 `--recovery_timeout` 设置恢复等待的时间。默认情况下恢复过程中的任何错误都会导致 slave 进程终止。如果配置 `--strict=false`，那么将会忽略恢复过程中发生的错误。为了开启框架检查点功能，`FrameworkInfo.checkpoint` 域需要在框架向master进行注册时被设置为 `true`。

从 Mesos 0.22 版本开始，slave 的检查点功能被默认开启，`--checkpoint` 选项也被从 slave 中删除。

检查点功能也使得状态更新变得更可靠。执行器驱动将所有发送给执行器的状态变化缓存起来，并在和重启后的slave重新建立连接时将状态变化发送给slave。这样所有的框架都可以透明地享用到到检查点带来的好处，执行器可以像正常情况一样发送消息。由于 master 是无状态的，并且所有信息只能从 slave 获取，所以如果没有检查点，在 master 故障恢复的过程中状态更新可能会丢失。在极端情况下，当 slave 重启时执行器退出，这会导致状态丢失。上一章介绍过，这种情况下状态一致性调解会辅助 slave 的恢复。

维护状态

Mesos 的维护状态允许在进行硬件升级或者维护任务（如操作系统升级等）时将 slave 从集群中移除。Mesos 的维护状态特征可以将 slave 将要进入维护状态的信息通知给框架，这样框架将不会在 slave 上启动新的任务。Mesos 等待 slave 上任务的结束并通知框架下一批将要进入维护状态的 slave，这样框架可以预先采取行动，使整个维护的过程更加流畅。

在本书写作时，维护状态的功能还处于开发中。

下面是 slave 进入维护状态的步骤：

1. 将要进入维护状态的 slave 会在它发送的资源 offer 中表明自己的意向。
2. Mesos master 停止将该 slave 的资源分配给框架。
3. Mesos 向在该 slave 运行任务的框架发送撤销资源 offer 的指令。
4. 框架接收到撤销指令，根据调度器的实现，可以将任务重新调度到其他节点。
5. slave 在指定时间内停止运行的所有任务。
6. slave 向 master 发送撤销注册事件，表明自己将要下线。这样 master 可以立刻知道 slave 节点已经消失，而不是等待超时。
7. slave 节点可以安全地进行升级维护。
8. slave 维护结束，再次向 master 发送资源 offer，重新成为集群的一部分。

一旦 slave 被调度进入维护状态，slave发送的资源 offer 将含有 `resource_expiry_time` 域，代表资源 offer 将在这个时间后失效。这些资源可以通过撤销 *offer*来请求，撤销 *offer*可以让框架在给定时间内退回指定资源。一旦 slave 上没有任何任务，master 会将这个 slave 标记为 deactivated。维护结束后，运维人员可以利用 `unschedule` 来通知 Mesos master，然后这个 slave 就可以继续为集群提供服务了。

这项工作目前还在进行中，计划通过下面的 HTTP 接入点来提供维护的基础工作。

- `/maintenance/schedule`：将 slave 调度为维护状态。下面是一个 POST 请求的示例，它要求 `slave1` 和 `slave2` 在特定时间被调度为维护状态。

```
{
  "schedule": [
    { "hostname" : "slave1", "time" : "2015-11-04T20:00:00" },
    { "hostname" : "slave2", "time" : "2015-11-04T22:00:00" },
  ]
}
```

- `/maintenance/unscheduled`：通知 slave 维护工作的完成。

```
{
  "unschedule": [
    { "hostname" : "slave1" },
    { "hostname" : "slave2" }
  ]
}
```

- `/maintenance/status`：获得当前的维护状态信息。

Mesos 接口

Mesos 的外部接口提供了其他工具与 Mesos 进行集成的方法。

Mesos REST 接口

Mesos 提供 REST 接口来和 Mesos 进行交互，可以通过浏览器或者命令行工具如 curl 来访问 REST API。下面的表格列举了 Mesos 所提供的 REST 服务接入点。所有的接入点默认都返回 JSON 格式结果。

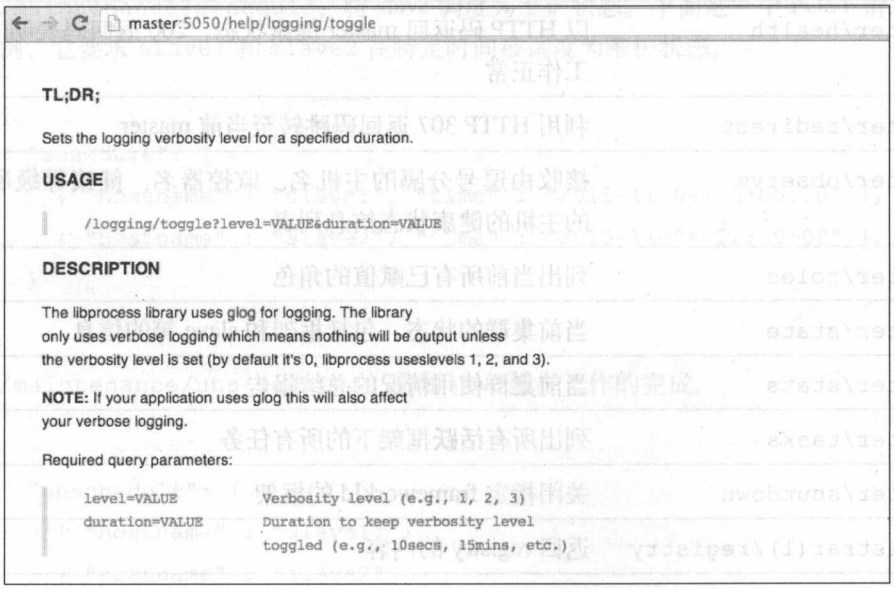
命令	用途
`/__processes__`	列出集群中的进程

`/files/browse`
`/files/debug`
`/files/download`
`/files/read`
	Mesos 文件 REST 接口，用来对文件进行浏览、读取等

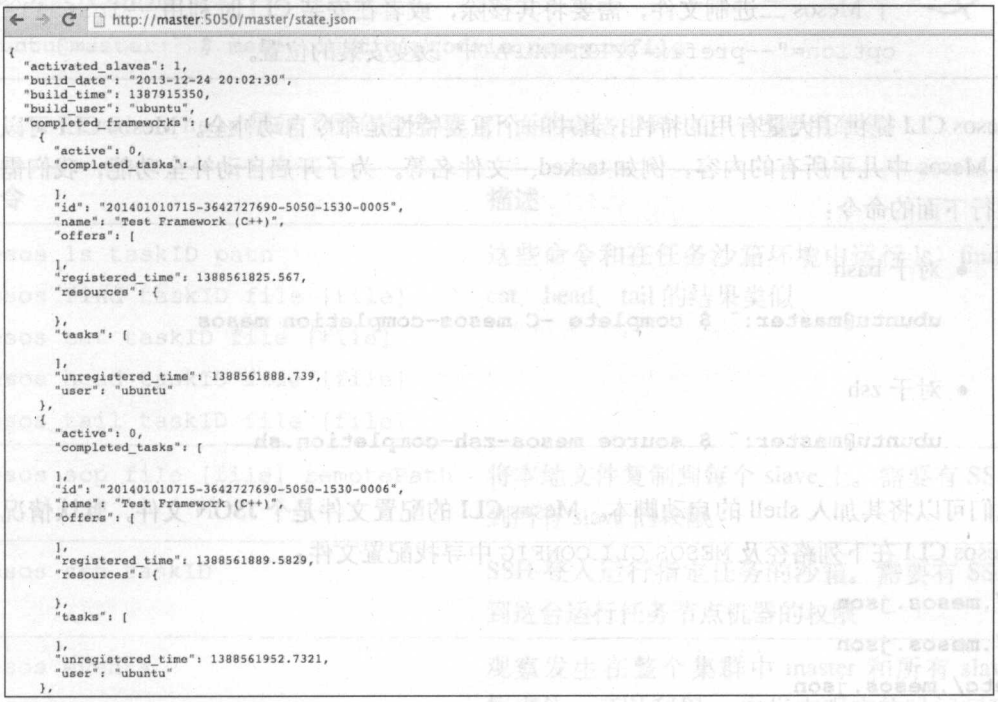
`logging/toggle`	在一段时间内开启日志模式
`/master/health`	以 HTTP 码返回 master 健康状态，200 返回码表明 master 工作正常
`/master/redirect`	利用 HTTP 307 返回码跳转至当前 master
`/master/observe`	接收由逗号分隔的主机名、监控器名、健康等级所代表的主机的健康状态信息列表
`/master/roles`	列出当前所有已赋值的角色
`/master/state`	当前集群的状态，包括框架和 slave 等的信息
`/master/stats`	当前集群使用情况的总结报告
`/master/tasks`	列出所有活跃框架下的所有任务
`/master/shutdown`	关闭指定 frameworkId 的框架
`/registrar(1)/registry`	返回 registry 的内容
`/profiler/start` `/profiler/stop`	开启/关闭 Mesos 剖析器
`/metric/snapshot`	提供当前度量指标的快照

Mesos 网页接口是在 REST API 之上构建的。所有的帮助页面自身也是接入点。可以通过`http://master:5050/help`来访问所有可用的 REST 接入点。

可以通过点击接入点的链接来获取更详细的帮助信息，例如可以通过点击`http://master:5050/help/logging/toggle/`获取 toggle 接入点的详细帮助文档。



例如，通过 `http://master:5050/master/state.json` 获取 master 的状态信息。



Mesos CLI

命令行接口是管理大规模集群的利器。Mesos 社区为 Mesos 开发了一个功能强大的 CLI 工具（`https://github.com/mesosphere/mesos-cli`）。

由于 Mesos 将集群中所有资源同等对待，资源没有特殊的标记，所以在排查生产系统的线上问题时，需要组合各个工具和接口方面的信息才能定位问题。举例来说我们想利用 SSH 来进入运行某个特殊任务的 slave 沙箱环境，或者想知道指定框架在哪些节点上运行任务，这样的需求很频繁但也很难完成。Mesos CLI 的目标是将这些日常的工作流组合起来。它以任务为中心进行设计，大部分的命令都接收 `taskID` 作为参数。智能匹配使得无须输入完整的 ID，只需部分 ID 就能获得匹配结果。CLI 遵循 UNIX 命令的行为风格，你可以用管道将其输入输出和其他命令组合起来使用。Mesos CLI 由 Python 编写，可以利用 `pip` 来安装 CLI：

```
ubuntu@master:~ $ pip install mesos.cli
```

命令的名称为 mesos，通常被安装在 /usr/local/bin。如果你的机器安装了 Mesos 二进制文件，需要将其移除，或者在安装 CLI 时利用 --install-option="--prefix=$PREFIX_PATH" 改变安装的位置。

Mesos CLI 提供了大量有用的特性，其中一个重要特性是命令自动补全。Mesos CLI 可以补全 Mesos 中几乎所有的内容，例如 tasked、文件名等。为了开启自动补全功能，我们需要运行下面的命令：

- 对于 bash

```
ubuntu@master:~ $ complete -C mesos-completion mesos
```

- 对于 zsh

```
ubuntu@master:~ $ source mesos-zsh-completion.sh
```

我们可以将其加入 shell 的启动脚本。Mesos CLI 的配置文件是个 JSON 文件。默认情况下 Mesos CLI 在下列路径及 MESOS_CLI_CONFIG 中寻找配置文件：

```
./.mesos.json
~/.mesos.json
/etc/.mesos.json
/usr/etc/.mesos.json
/usr/local/etc/.mesos.json
```

下面是一个配置文件的样例：

```
{
  "profile": "default",
  "default": {
    "master": "zk://localhost:2181/mesos",
    "log_level": "warning",
    "log_file": "/tmp/mesos-cli.log",
    // http or https. Scheme to use to connect to Mesos
    "scheme": "http"
  }
}
```

使用 Mesos CLI 并不需要在本地安装 Mesos，可以在命令中指定 Mesos 的位置：

```
ubuntu@master:~ $ mesos config master zk://localhost:2181/mesos
```

CLI 支持 profile，这使得通过 CLI 管理多个 Mesos 集群变得十分方便。只需要一条命令就可以更改使用的配置文件：

```
ubuntu@master:~ $ mesos config profile new-profile
```

`mesos [cmd] --help` 展示了所有的选项。下面的表格中列出了一些常用命令：

命令	描述

`mesos ls taskID path`
`mesos find taskID file [file]`
`mesos cat taskID file [file]`
`mesos head taskID file [file]`
`mesos tail taskID file [file]`
	这些命令和在任务沙箱环境中运行 ls、find、cat、head、tail 的结果类似

`mesos scp file [file] remotePath`	将本地文件复制到每个 slave 上。需要有 SSH 到所有 slave 的权限
`mesos ssh taskID`	SSH 登入运行指定任务的沙箱。需要有 SSH 到这台运行任务节点机器的权限
`mesos events`	观察发生在整个集群中 master 和所有 slave 的事件。可以利用 -s 来指定观察的时间间隔（默认为 5 秒）
`mesos config [key] [value]`	输出 Mesos CLI 的 JSON 格式配置。如果只有 key 被指定，那么它将返回对应的值。如果 key 和 value 都被指定，那么它将会设置这个值。key 可以是任何有效的配置参数
`mesos resolve [master-config]`	返回当前主 master 的地址
`mesos state [slave]`	返回 master 或者匹配的 slave 的完整 JSON 格式状态信息

所有子命令都是一个单独的脚本，可以直接通过 `mesos` 前缀来调用各个子命令，例如 `mesos cat` 和 `mesos-cat` 是等价的。`mesos` 脚本扫描 `PATH` 路径并将所有 `mesos` 为前缀的文件作为自己的子命令。也可以选择自己的语言来实现子命令，只需要按照 `mesos-somecmd` 的格式进行命名并放入 `PATH` 路径即可。`mesos-config`、`mesos-resolve` 和 `mesos-state` 返回了可迭代的结构，可以利用它们来构建新的命令。

配置

本节将列举所有的 Mesos 配置参数、描述及默认值。配置参数的最新列表维护在 `http://mesos.apache.org/documentation/latest/configuration` 处。所有的配置参数首先从环境变量中搜索，再从命令行中解析。环境变量名比配置参数多了 `MESOS_` 前缀，并且全部字母大写。例如，命令行参数的 `--cluster` 对应环境变量中的 `MESOS_CLUSTER`。时间相关的配置值的单位均为用户友好的单位，例如秒、分钟、小时等。

下面是 master 和 slave 所共有的配置项：

Mesos slave 和 Mesos master 共有的配置项

配置项	描述	默认值
`--ip`	master 所绑定的 IP 地址。当机器上有多个网卡时，推荐使用该配置参数	默认接口 IP
`--port`	监听端口	`5050`
`--log_dir`	日志文件路径	没有默认值，不指定 log_dir 的情况下不会打印日志
`--logbufsecs`	log 刷新间隔	0
`--logging_level`	等于或者高于这个级别的日志将会被打印。可能的级别有 `INFO`、`WARNING` 和 `ERROR`	`INFO`
`--external_log_file`	指定外部管理日志文件，能够被网页界面和 HTTP API 访问。如果不使用这种方式，Mesos将无法获知使用 stderr 方式打印的日志	
`--hooks`	由逗号分隔的 hook 模块列表，这些模块将被安装在 master 上	
`--modules`	加载入内部子系统的模块列表。后面可以接一个 JSON 字符串或者包含 JSON 字符串的文件路径	
`--hostname`	master 或者 slave 对外的主机名。如果未被设置则使用系统主机名	

（续表）

配置项	描述	默认值
--quiet 或者 --no-quiet	控制日志行为 --quiet 关闭向标准错误输出日志	false
--version	显示 Mesos 版本	

Mesos master

下面的列表展示了 Mesos master 的配置选项，可以通过 mesos-master --help 命令来获得。

Mesos master 配置项

配置项	描述	默认值
--quorum	当使用基于 replicated log 的 registry 时所需的 quorum 个数。这个数字决定了至少有多少 master 同意状态才可以写入 registry。这个数字应该大于 master 个数的一半	
--work_dir	所有框架的工作路径，以及 replication log 存放的路径。这是一个必填的配置参数	
--zk	为集群提供选主服务的 ZooKeeper URL，高可用部署的必填配置参数	
--zk_session_timeout	ZooKeeper 会话超时时间	10 秒
--log_auto_initialize	检查是否自动初始化 replicated log。如果置为 false，replicated log 在使用前必须手动初始化	true
--acls	ACL 的 JSON 配置文件。后接的值可以是一个JSON 字符串，也可以是一个包含 JSON 字符串的文件	

（续表）

配置项	描述	默认值
--allocation_interval	两次批量分配的间隔时间	1 秒
--authenticate	检查没有通过鉴权的框架是否允许注册。如果设置为 true，未经鉴权的框架将不允许注册	false
--authenticate_slaves	检查没有通过鉴权的 slave 是否允许注册。如果设置为 true，未经鉴权的 slave 将不允许注册	false
--authenticators	框架和 slave 所使用的鉴权实现方式。可以通过 modules 来指定另一个模块	crammd5（挑战应答认证机制——消息摘要算法）
--credential	包含证书信息的 JSON 文件路径	
--cluster	Mesos 展示在网页接口中的名字	
--framework_sorter	为每个框架分配资源的策略	drf
--user_sorter	为每个用户分配资源的策略	drf
--offer_timeout	在这个时间后，资源将从框架被回收。这可以处理一些框架接收资源配后一直占有不释放的问题	
--rate_limits	速率限制的 JSON 配置，后接的值可以是一个 JSON 字符串，也可以是一个包含 JSON 字符串的文件	
--recovery_slave_removal_limit	故障恢复时可移除的 slave 最大比例	100%
--slave_removal_rate	master 移除 slave 的最大速率，格式为 slave 数量/时间间隔	没有配置
--registry	registry 所使用的持久化策略，可选值为 replicated_log 或者 in_memory	replicated_log

（续表）

配置项	描述	默认值
--registry_fetch_timeout 和 --registry_store_timeout	当 fetch 或者 store 失败后再次发起 fetch 和 store 的时间间隔	fetch 为 1 分钟，store 为 5 秒
--registry 或者 --no-registry_strict	目前是一个实验性的配置项。如果设置 --no-registry_strict，那么 registrar 不会拒绝任何 slave 的注册重新注册以及移除。这可以用来引导一个已运行的集群	--no-registry_strict
--roles	框架在集群中的角色信息。配置值为逗号分隔的一个列表，例如 --roles="test,prod"	
--root_submissions 或者 --no-root_submission	是否允许 root 提交框架	true
--slave_reregister_timeout	master 重新选举后 slave 重新注册的超时时间。如果 slave 没有重新注册，则它将会被从 registry 中删除，和 master 之间的通信也会被关闭，目前的默认值为 10 分钟	10 分钟
--webui_dir	Web UI 文件的目录路径	
--weights	由逗号分隔的 role=weight 对列表，代表了每个角色的优先级。每个用户的支配资源需要和权重相除。权重为 2 的用户要比权重为 1 的用户多获得一倍的资源。在一个角色中，离加权 fair 最远的框架就是给定的资源。例如，--weights="test=1,prod=2"	

（续表）

配置项	描述	默认值
--whitelist	白名单配置文件路径，包含可接受资源 offer 的 slave	默认情况下没有白名单，所有 slave 的资源 offer 都是被接受的
--resource_monitoring_interval	执行器资源使用监控间隔	1秒
--max_executors_per_slave	每个 slave 上所能运行的执行器最大个数。用于网络隔离器	

Mesos slave

下面的列表展示了 Mesos slave 的配置选项，可以通过 `mesos-slave --help` 命令来获得。

Mesos slave 的配置选项

配置项	描述	默认值
--attributes	节点所包含的属性列表，属性之间以逗号分隔，调度器可以利用属性作为调度的限制条件。属性可以被当作格式为 "name:value" 的标签，例如：--attributes="rack:r1, special_hardware:1, attr1:value1"	没有属性

（续表）

配置项	描述	默认值
--master	指定要连接的 master 路径。这个值可以是主机名、IP 地址、一个或多个 ZooKeeper master URI。也可以将前两者之一存入文件，配置指向文件路径。这是一个必填的选项。例如：--master=localhost:5050 --master=10.0.0.10:5050,10.0.0.20:5050 --master=zk://user:password@10.0.0.11:2181/mesos --master=/etc/mesos/zk	
--checkpoint 或者 --no-checkpoint	检查是否将 slave 和框架的信息以检查点形式保存到磁盘。检查点允许 slave 在重启后进行恢复	true
--work_dir	框架工作路径	/tmp/mesos
--recover	指明恢复策略，可以是重连接或者清理。重连接意味着 slave 会和之前仍在运行的执行器重新连接。这只有在 --checkpoint 选项存在的情况下才会生效。否则 slave 会以新 slave 的身份向 master 注册。清理策略意味着所有在 slave 关闭前运行的执行器都会被杀死	重连接
--strict 或者 --no-strict	检查恢复过程中的错误是否被当成致命错误。如果设置为 true，所有恢复中的错误都被认为是致命的。否则所有的错误都被忽略，slave 可以尽可能多地恢复到之前的状态	true
--recovery_timeout	等待 slave 恢复的超时时间。过了超时时间后，所有失去和 slave 连接的执行器将会自杀	15 分钟

（续表）

配置项	描述	默认值
--isolation	使用的隔离机制。可选的值有posix/cpu,posix/mem 。如果要使用 cgroups 隔离器，需要在这里设置 "cgroups/cpu,cgroups/mem"	posix/cpu,posix/mem
--cgroups_enable_cfs 或者 -no-cgroups_enable_cfs	开启 CFS 带宽限制的特性对 CPU 进行硬限制	false
--cgroups_hierarchy	cgroups 的根路径	/sys/fs/cgroup
--cgroups_limit_swap	开启内存和交换区的限制	false
--cgroups_root	根 cgroup 的名字	mesos
--containerizers	用逗号分隔的列表来指明 Mesos 所使用的容器机实现。目前支持的容器机有 Mesos、External 和 Docker。Mesos 会按照列表的顺序进行尝试。例如 containerizers="mesos,docker"	mesos
--containerizer_path	指定外部容器机的可执行文件，该选项需要和 --isolation=external 一起发挥作用	
--default_container_image	外部容器及所使用的默认镜像（任务没有指定镜像），这个选项只在 --isolation=external 的情况下生效。例如：--default_container_image=docker:///libmesos/Ubuntu:13.10	
--default_container_info	在 ExecutorInfo 中未显式指定 ContainerInfo 时使用的默认 ContainerInfo。该值为 JSON 格式	
--docker	Docker 可执行文件的位置	docker

（续表）

配置项	描述	默认值
--docker_remove_delay	在删除 Docker 容器前的等待时间	6 小时
--docker_sandbox_directory	沙箱目录映射到容器内目录的路径	/mnt/mesos/sandbox
--docker_stop_timeout	Docker 停止实例的超时时间，如果实例在这段时间内没有停止，则它将会被杀死	0 秒
--resources	slave 为每个角色所提供的资源。我们可以限制每个角色所能使用资源。格式为："resource(role):value"。例如：--resources="cpus(prod):16;cpus(test):2;disk(*):"	
--default_role	在 --resources 中没有指定角色的配置项将使用的默认角色。所有被自动检测到且不在 --resources 中的资源也会分配给这个角色	"*"所有角色都有访问资源的权限
--disk_watch_interval	周期性检查磁盘使用率的时间间隔	1 分钟
--container_disk_watch_interval	周期性检查容器磁盘配额的时间间隔，posix/disk 隔离器使用这个配置	15 秒
--enforce_container_disk_quota 或 --no-enforce_container_disk_quota	开启容器磁盘配额功能。posix/disk 隔离器使用这个配置	false
--executor_registration_timeout	执行器和 slave 注册的超时时间	1 分钟
--executor_shutdown_grace_period	执行器关闭的时间	5 秒

（续表）

配置项	描述	默认值
--frameworks_home	附加在执行器相对 URI 前的路径	
--gc_delay	执行器工作目录在垃圾回收前所能存活的最长时间。根据磁盘使用率的情况，垃圾回收可以能在这个时间之前发生	1 周
--gc_disk_headroom	这个值用来调整执行器目录回收的时间。每隔 --disk_watch_interval 时间，目录的可存活时间按照下面的公式进行更新：gc_delay * max(0.0, (1.0 - gc_disk_headroom - disk usage))	0.1
--hadoop_home	指向 HADOOP_HOME 的路径，用来从 HDFS 上获取执行器。目前没有默认值	
--launcher_dir	Mesos 二进制文件的位置	/usr/local/ lib/mesos
--registration_backoff_factor	如果所有 slave 同时尝试向 master 进行注册，将会导致网络风暴。为了避免风暴的发生，每个 slave 在启动时随机会等待 [0,registration_backoff_factor] 之间的任一时长后再向 master 发起连接。重试等待事件的次数会指数上升（第 *n* 次重试等待的时间范围为[0,registration_backoff_factor *2^n]，最长为 1 分钟）	1秒
--credential	包含证书信息 JSON 文件路径	
--authenticatee	框架和 slave 所使用的鉴权实现方式。可以通过 modules 来指定另一个模块	crammd5（挑战应答认证机制——消息摘要算法）
--perf_interval	性能状态采样的时间间隔	1 分钟
--perf_duration	每次性能状态采样的时间，这个值必须小于 perf_interval	10 秒

（续表）

配置项	描述	默认值
--perf_events	perf_event 隔离器使用的容器性能采样事件，由逗号进行分隔。事件名称在发送给 PerfStatistics 时会被进行了标准化，所有字母小写，连字符被替换为下画线。例如 cpu-cycles 会被替换为 cpu_cycles。隔离器一系列的性能事件可以通过 perf list 命令查看	
--resource_monitoring_interval	执行器资源使用监控的时间间隔	1 秒
--slave_subsystems	slave 运行时的 cgroup 子系统，包括 memory、cpuacct 等，由逗号进行分隔。目前的版本并没有对资源的使用进行限制，只是利用 cgroup 进行监控。它们继承于 mesos root cgroup	
--switch_user 或 --no-switch_user	检查是以提交任务的用户身份还是 slave 运行的身份执行任务	true
--ephemeral_ports_per_container	网络隔离器分配给每个容器临时端口的数量。这个值必须是 2 的幂次	1024
--eth0_name --lo_name	公共网络接口和环回网络接口的名称。如果没有指定，网络隔离器会去猜测它们的值	
--egress_rate_limit_per_container	限制容器的出口网络流量，单位为字节/秒。如果没有指定或者值为 0，网络隔离器不会对出口流量做任何限制	
--network_enable_socket_statistics	检查是否收集每个容器的网络统计信息	false

Mesos 构建选项

当从源码构建 Mesos 时，下面的选项可以用来控制 Mesos 的构建，以适应不同的生产环

境。所有的配置项都可以通过 `configure --help` 来浏览。

下面列出了一些 Mesos 的特色构建选项：

配置项	描述	默认值
`--enable-shared`	检查是否构建共享库	`true`
`--enable-static`	检查是否构建静态库	`true`
`--enable-fast-install`	检查是否为快速安装进行优化	`true`
`--disable-libtool-lock`	关闭 libtool 锁	`false`
`--disable-java` `--disable-python`	关闭 Java 或者 Python 绑定	`false`
`--enable-debug` `--enable-optimize`	检查是否开启调试或者优化选项	`false`
`--disable-bundled`	检查是否利用已安装的依赖而不是使用 Mesos自带的依赖	`false`

`--disable-bundled-distribute`
`--disable-bundled-pip`
`--disable-bundled-wheel`
	检查是否分别排除了 `istribute`、`pip` 和 `wheel` 包的创建和使用，而不是使用 PYTHONPATH 下的安装版本	`false`

`--disable-python-dependency-install`	检查在 `make install` 的过程中是否安装 Python 依赖	`false`

下面是打包相关的选项：

配置项	描述	默认值
`--with-gnu-ld`	C 编译器使用 GNU 的连接器	No

`--with-zookeeper`
`--with-leveldb`
`--with-glog`
`--with-protobuf`
`--with-gmock`
	检查是否排除了指定位置的 ZooKeeper、LevelDB、glog、protocol buffer 和 gmock 的构建和使用，转而从源码中编译	No

（续表）

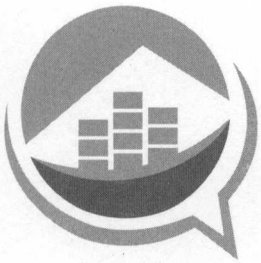
配置项	描述	默认值
--with-curl --with-sasl --with-zlib --with-apr --with-svn	指定所使用的 CURL、SASL、ZLib、APR 和 SVN 包的位置	
--with-sysroot	在指定路径寻找相关依赖库	
--with-network-isolator	检查是否构建网络隔离器	false

除了上面的选项，下面的环境变量也会影响构建的结果。当这些包没有安装在标准位置时，这些值是非常有用的。

环境变量	描述
JAVA_HOME	JDK 安装路径
JAVA_CPPFLAGS	**Java 本地接口（JNI）**的预处理选项
JAVA_JVM_LIBRARY	libjvm.so 路径
MAVEN_HOME	maven 安装路径
PROTOBUF_JAR	protocol buffer jar 包绝对路径
PYTHON	Python 解释器路径
PYTHON_VERSION	Python 版本

小结

本章介绍了 Mesos 需要为生产环境所提供的各种功能，讨论了一些工具的使用以及管理 Mesos 集群的最佳实践。本书的内容也到此结束，让我们开始搭建可扩展的应用吧!



DockOne.io

Community of Container

DockOne.io 成立于 2014 年，是国内最大的容器社区。社区主要关注 Docker、Mesos、CoreOS、Kubernetes、Ceph、OpenStack 等容器生态圈相关软件，致力于为广大容器爱好者提供一个分享、学习和交流的平台，目前已有活跃会员逾 50000 精品文章 1000 余篇。

Broadview
www.broadview.com.cn
博文视点 · IT出版旗舰品牌
技术凝聚实力 · 专业创新出版

Mesos

大数据资源调度与大规模容器运行最佳实践

Apache Mesos是集群管理器，它能够为分布式的应用程序或框架提供高效的资源隔离和共享。它允许开发人员在一个动态共享节点池上并行运行Hadoop、Spark、Storm和其他应用程序。有了Mesos，你就可以在多租户环境里管理各种资源了。

本书从基础入手，逐步介绍了Mesos提供的所有功能。首先介绍了如何在不同的环境（从数据中心到云环境）中搭建Mesos。然后讲述了如何在Mesos上使用多种服务调度器（如Chronos、Aurora和Marathon）来实现自管理的平台即服务环境。接着深入研究了Mesos的基础，以及如何使用Mesos构建分布式应用程序。

最后介绍了Mesos的运维细节，包括日志、监控、高可用和恢复。

能从书中学到什么

◎能够在数据中心或者云环境中搭建Mesos集群。

◎能够在Mesos上使用Hadoop、Spark和Storm等框架进行数据分析。

◎熟悉如何在Mesos上使用Marathon、Chronos和Aurora来管理服务。

◎深入理解如何使用Mesos API编写分布式应用程序。

◎学会如何自动化并管理Mesos集群，以及其他运维技能，比如日志和监控。

◎深入理解Mesos的基础和内在工作机制。

本书的目标读者

本书写给想要使用Apache Mesos构建并运行可扩展且自动容错的应用程序的开发人员和运维人员。本书要求读者具备基本的编程知识和Linux基础。

业内专家力荐

很高兴看到国内这么快就有了关于Mesos的中文书籍。本书由浅入深，循序渐进，从应用案例到架构实现面面俱到，推荐想了解Mesos的同学阅读！

郭蕾 InfoQ主编

本书深入浅出地介绍了Mesos的基本原理、API 框架及多种典型的解决方案，既有原理，又有应用案例。无论是站在服务开发者还是服务运维者的角度，读过此书后都会有一种醍醐灌顶的感觉，对Mesos有一种全新的认识，在服务管理、构建和运维上获得新的思路。

谢广军 百度高级技术经理，云计算技术服务人

由DockOne社区翻译的《Mesos：大数据资源调度与大规模容器运行最佳实践》一书，作为国内第一本关于Mesos的书籍，从实践入手，将带你深入浅出地认识Mesos，对国内Mesos实践者来说，是很好的入门读物。

左玥 灵雀云创始人兼CEO

这是一本理论与实践相结合的实践性书籍，我相信无论是新手或是资深用户，都可以从中获得所需要的知识和技能。

汪洋 华为开源能力中心开源战略规划专家

如果你想了解Mesos的一切，如果你想与硅谷最新技术同行，请与我们一起认真阅读这本佳作。

涂彦 腾讯游戏运维总监

上架建议：大数据

[PACKT] PUBLISHING

ISBN 978-7-121-26902-8

9 787121 269028 >

定价：65.00元

策划编辑：张春雨
责任编辑：白 涛
封面设计：李 玲